Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования «Ивановский государственный
энергетический университет имени В.И. Ленина»

На правах рукописи

Чадов Сергей Николаевич

Разработка и исследование высокопроизводительного программного комплекса для решения жестких систем ОДУ на графических процессорах общего назначения

Специальность 05.13.18 —
Математическое моделирование, численные методы и комплексы программ

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
д-р физ.-мат. наук, проф.
Ф.Н. Ясинский

Иваново – 2014

Содержание

Введение

Актуальность темы диссертации.

Математическое и численное моделирование является одним из основных инструментов анализа технологических процессов, протекающих в сложных технических устройствах и системах. Результаты численных экспериментов играют важную роль как на стадии проектирования устройств, так и при оптимизации режимов их работы. Математические модели указанных процессов в отдельных устройствах технической системы часто базируются на системах обыкновенных дифференциальных уравнений (ОДУ), которые могут быть решены только численно. До середины 90-х годов XX века практически единственным инструментом для решения жестких систем были неявные методы, в частности, основанные на так называемых формулах дифференцирования назад и развитые У. Гиром. В настоящее время эти методы также используются, современные реализации можно найти, например, в пакете SUNDIALS [44], однако их область применения сужается по причине развития теории и практических реализаций специализированных явных методов. Среди работ в этом направлении можно отметить в первую очередь работы У. Гира и его соавторов по проекционным методам [38, 47], а также Я. Вервера [61], А. Абдулла [20], А.А. Медовикова [51], Л.М. Скворцова [8, 9] и других авторов. Явные методы имеют несомненные преимущества по производительности по сравнению с неявными ввиду отсутствия необходимости решения на каждом шаге системы, как правило, нелинейных, алгебраических уравнений, существенно замедляющего вычисления. В связи с этим использование явных методов часто дает большее преимущество, чем распараллеливание неявных методов и использование более производительного оборудования. С другой стороны, неявные методы, используемые для решения жестких систем, обладают важными свойствами устойчивости (A-устойчивости, L-устойчивости и т.д), в результате чего полученное ими решение является гарантированно устойчивым. Явные методы в большинстве случаев не обладают такими свойствами. Соответственно, явные методы могут быть успешно применены не ко всем жестким системам, причем возможность применения к какой-либо конкретной системе зависит от нетривиальных характеристик спектра систе-

мы, вычисление которых может быть довольно сложно и часто заменяется постановкой численного эксперимента.

Несмотря на быстрый рост возможностей вычислительной техники, при моделировании процессов, протекающих в сложных системах, имеет место дефицит вычислительной мощности. Одним из способов повышения вычислительной мощности является передача части вычислений на специализированные вычислительные устройства, способные решать некоторые задачи существенно эффективнее вычислительных устройств общего назначения. Современным примером таких специализированных вычислительных устройств являются графические процессоры общего назначения (GPGPU), имеющие на данный момент при относительно низкой цене теоретическую пиковую производительность порядка 4.5 Tflops, тогда как современный центральный процессор имеет пиковую производительность порядка 200 Gflops [52]. При этом получаемое ускорение является следствием архитектуры графических процессоров, которая не позволяет решать любые задачи столь же эффективно, как центральный процессор, и часто требует особых приемов программирования. Также нужно отметить, что использование данных вычислительных устройств – довольно молодое направление в науке, и по этой причине количество уже созданных стандартных программных средств для решения тех или иных задач относительно невелико. Следовательно, реализация высокопроизводительных средств моделирования на графических процессорах общего назначения является актуальной задачей.

Использование графических процессоров обладает рядом преимуществ:

–Моделирование с использованием графических процессоров, как правило, может осуществляться быстрее физического течения моделируемого процесса, что позволяет прогнозировать наступление различных опасных ситуаций, а также рассчитывать эффект тех или иных управляющих воздействий.

– Моделирование на графических процессорах позволяет исследовать системы со множеством различных значений параметров и выбирать оптимальные на основании полученных результатов.

– Стоимость современных графических процессоров сравнима со стоимостью центральных процессоров персональных компьютеров при существенном повышении производительности. В некоторых случаях возможно использование систем из двух или четырех графических процессоров, позволяющих

решать сложные задачи, для которых обычно используются суперкомпьютеры, имеющие на порядок большую стоимость.

Поэтому целью данной работы является разработка программного комплекса на основе высокопроизводительного метода моделирования технических систем с использованием графических процессоров, включающего в себя выбор алгоритма, его программную реализацию и применение этого метода к конкретной модельной системе.

Задачи исследования:

– Построение системы ОДУ, предназначенной для моделирования некоторого класса технических систем. Исследование ее свойств, выяснение применимости и эффективности различных методов интегрирования, как общих, так и специализированных.

– Выбор оптимального метода, обеспечивающего при сохранении приемлемого уровня точности минимального времени вычислений.

– Разработка высокопроизводительной программы интегрирования полученной системы ОДУ с использованием графических процессоров общего назначения. Обоснование преимуществ разработанной реализации перед решением на центральном процессоре.

Решение поставленных задач осуществлялось методами теории решения жестких систем ОДУ, в частности проекционными методами, а также методами теории параллельного программирования. С методологической точки зрения основным методом исследования являлась постановка вычислительного эксперимента.

Научная новизна результатов работы*:*

В области математического моделирования:

– Предложено семейство систем обыкновенных дифференциальных уравнений, позволяющее моделировать широкий класс различных технических систем, обладающих рядом общих свойств и сводимых друг к другу с использованием принципа динамических аналогий.

В области численных методов:

– Показано, что свойства систем дифференциальных уравнений из этого семейства определяются их размерностью. Так, система уравнений для одного источника является нежесткой, тогда как система уравнений для несколь-

ких источников является жесткой, что требует изменения подхода к ее численному решению.

– Показана возможность решения данной жесткой системы дифференциальных уравнений специализированными явными методами, а также преимущество такого решения перед стандартными неявными методами.

В области комплексов программ:

– Предложен способ реализации интегрирования рассмотренной системы уравнений на специализированном вычислительном устройстве - графическом процессоре общего назначения. Показано, что такая реализация значительно (в 10 и более раз) превосходит аналогичную реализацию на центральном процессоре.

Достоверность основных научных положений и выводов работы подтверждается соответствием полученных решений для тестовых задач результатам расчета широко распространенными и заведомо корректными методами. Корректность созданных программных библиотек обеспечивается, в частности, набором тестов и системой автоматизированного тестирования. Обоснованность научных результатов подтверждается корректным использованием методов решения систем ОДУ.

Практическая ценность работы заключается в следующем:

– Разработана программная реализация решения жестких систем ОДУ на графическом процессоре общего назначения, которая способна стать частью САПР, АСУ ТП, либо другой подобной системы.

– Создана библиотека программ для решения систем ОДУ, в том числе с использованием графических процессоров общего назначения, которая может быть применена к широкому кругу задач моделирования различных технических процессов.

Основные результаты работы были доложены, обсуждены и получили одобрение на следующих научных конференциях: Международной НТК «Состояние и перспективы развития электротехнологии (XV Бенардосовские чтения)». – Иваново, ИГЭУ, 2009; Региональных НТК «Применение многопроцессорных суперкомпьютеров в исследованиях, наукоемких технологиях и учебной работе». – Иваново, ИГТА, 2007, 2008. По материалам диссертации опубликовано 14 печатных работ, в том числе 9 статей в изданиях, предусмотренных перечнем ВАК, свидетельство о регистрации программы для ЭВМ.

1. Решение ОДУ

1.1. Методы решения ОДУ

1.1.1. Основные определения

Обыкновенным дифференциальным уравнением (или системой обыкновенных дифференциальных уравнений, в данной работе эти термины являются взаимозаменяемыми) будем называть уравнение вида

$$\mathrm{F(t,y,}\frac{dy}{dt}\mathrm{,}\frac{d^2y}{dt^2}\mathrm{,...)=0,}$$

где t – независимая переменная (время); F и y−некоторые вектор-функции. Уравнения вида

$$\frac{dy^n}{dt^n}\mathrm{=f(t,y,}\frac{dy}{dt}\mathrm{,}\frac{d^2y}{dt^2}\mathrm{,...)}$$

называются разрешенными относительно старшей производной [1], и в дальнейшем, если не указано обратное, будем считать, что исходное уравнение представлено в такой форме. Число n, являющееся максимальным порядком производной, участвующей в уравнении, называется порядком уравнения [1]. Большинство численных методов (включая все, рассматриваемые в данной работе) предназначены для решения уравнений первого порядка. Системы уравнений высших порядков приводятся к необходимому виду добавлением дополнительных уравнений вида $\frac{dy}{dy} = \hat{y_1}, \frac{d\hat{y_1}}{dt} = \hat{y_2}, ...$ и соответствующей заменой переменных [63].

Решением задачи Коши для обыкновенного дифференциального уравнения является нахождение функции y или, при решении, как в нашем случае, численными методами, последовательности ее значений при заданных начальных условиях y_0 [63].

Система называется автономной, если функция f не зависит от времени [63]. Как будет показано далее, для автономных систем некоторые методы предполагают более простое выражение для вычислений.

1.1.2. Общие методы

Существует множество широко известных алгоритмов численного решения систем ОДУ, характеризующихся различным порядком точности и скоростью работы. Можно выделить несколько основных групп распространенных методов [63]:

- *Явные методы группы Рунге–Кутты.* Сюда относятся простейший явный метод Эйлера [19] и его модификации [63], а также широко известный метод Рунге-Кутты 4-го порядка [65]. К этой же группе методов относится применяемый по-умолчанию в современных математических пакетах метод Дормана-Принса [33].
- *Неявные методы группы Рунге–Кутты*, например неявный метод Эйлера или метод трапеций [63]. Эти методы характеризуются существенно большими вычислительными затратами ввиду необходимости на каждом шаге решения системы алгебраических уравнений, однако, имеют преимущество в устойчивости.
- *Линейные многошаговые методы*, в частности, методы Адамса [27, 63].

В данной работе мы уделим большее внимание способам решения систем, которые в силу тех или иных причин плохо поддаются решению распространенными методами. Трудности могут быть вызваны многими факторами, наиболее существенными из которых, на наш взгляд, являются высокая размерность, жесткость системы и в меньшей степени высокие требования к точности.

1.1.3. Жесткие системы

При решении систем дифференциальных уравнений, возникающих во многих практических задачах в таких областях, как химическая кинетика, теория ядерных реакторов, теория автоматического управления, электротехника, электроника и т.д., часто возникает следующее явление: несмотря на медленное изменение некоторых (иногда – большей части) искомых функций, расчет приходится вести с очень малым шагом. Системы уравнений, обладающие таким свойством, получили название жестких. Дать строгое определение жесткой системы достаточно сложно. В литературе обычно используют

число жесткости, характеризующее степень жесткости системы. Число жесткости определяется как

$$g = \frac{\max\limits_i |Re\lambda_i|}{\min\limits_i |Re\lambda_I|},$$

где – λ_i собственные числа матрицы Якоби для правых частей системы дифференциальных уравнений [46]. Систему уравнений будем считать жесткой, если для нее $g \gg 1$. Например, для системы уравнений

$$\begin{cases} x' = -10000x - 4999y, \\ y' = -10001x - 5000y \end{cases}$$

$g \approx 44989$. Таким образом, эту систему можно считать достаточно жесткой, хотя стоит отметить, что четкой границы между жесткими и нежесткими системами не существует. По этой причине некоторые авторы (например, [32, 37, 46, 57, 58]) используют другое определение жесткости, которое мы будем называть «практическим» (а определение на основе значения числа жесткости – «теоретическим»). Систему уравнений будем называть жесткой, если при ее решении значение шага по времени ограничивается сверху устойчивостью решения. Для нежестких же систем значение шага по времени ограничивается в первую очередь желаемой точностью решения. Стоит отметить, что в общем случае жесткость системы (какое бы определение мы не выбрали) является локальной характеристикой системы и может меняться с течением времени.

1.1.4. Примеры жестких систем

Приведем несколько примеров жестких задач, встречающихся на практике. Во многих приложениях, в частности в электронике при анализе электрических цепей, в биологии при анализе электрической активности нейронов, в сейсмологии и т.д., возникают колебания, описываемые уравнением Ван-дер-Поля [45]:

$$\begin{cases} \dfrac{dy_1}{dt} = y_2, \\ \dfrac{dy_2}{dt} = E((1 - y_1^2)y_2 - y_1). \end{cases}$$

Жесткость данной системы зависит от параметра E, который на практике может принимать значения порядка нескольких миллионов, что делает эту систему очень жесткой. Другая хорошо известная задача представляет собой модель, описывающую реакцию Белоусова–Жаботинского [5] – реакцию, протекающую в колебательном режиме, при котором некоторые параметры реакции изменяются периодически. Существует несколько моделей, описывающих эту реакцию. Одна из них, так называемый орегонатор [36], представляет собой следующую систему уравнений (коэффициенты приведены для реакции с участием HBrO2 Br- и Ce):

$$\begin{cases} \dfrac{dy_1}{dt} = 77.27(y_2 + y_1(1 - 8.375{\cdot}10^{-6}y_1 - y_2)), \\ \frac{dy_2}{dt} = \frac{1}{77.27}(y_3 - (1 + y_1)y_2), \\ \frac{dy_3}{dt} = 0.161(y_3 - y_1). \end{cases}$$

Другая модель реакции Белоусова–Жаботинского – брюсселятор [55]:

$$\begin{cases} \frac{dy_1}{dt} = A + y_1^2 y_2 - (B + 1)y_1, \\ \frac{dy_2}{dt} = By_1 - y_1^2 y_2. \end{cases}$$

1.1.5. Устойчивость методов численного решения ОДУ

Численное решение жестких систем представляет собой определенную проблему. В первую очередь, эта проблема связана с устойчивостью методов численного решения дифференциальных уравнений. Устойчивостью называется свойство метода сохранять погрешность вычислений ограниченной с увеличением числа шагов [63].

Устойчивость методов численного интегрирования обычно исследуется на модельном уравнении

$$y' = \lambda y, \quad \lambda \in C, \quad Re(\lambda) \leq 0.$$

Рассмотрим, например, явный метод Эйлера:

$$y_n = y_{n-1} + h \cdot y'_{n-1.}$$

Для краткости, обозначим $z = h\lambda$, тогда для модельного уравнения эта формула примет вид

$$y_n = y_{n-1} + zy_{n-1.}$$

Погрешность вычислений $e_n = y_n - y(t_n)$ для данного метода будет равна (без учета ошибки округления)

$$e_n = (1 + z)e_{n-1.}$$

Таким образом, для выполнения условия устойчивости, т. е. ограниченности e при увеличении n, необходимо, чтобы

$$|1 + z| \leq 1,$$

откуда следует, что метод является устойчивым при z $\leq$ 2, а область его устойчивости (рисунок 1.1) представляет собой круг на комплексной плоскости радиуса 1 с центром в точке (-1,0) [4].

Таким образом, мы видим, что устойчивость ограничивает выбор размера шага интегрирования сверху. Для жестких систем это приводит к тому, что максимальное значение шага, при котором метод сохраняет устойчивость, может являться неоправданно малым.

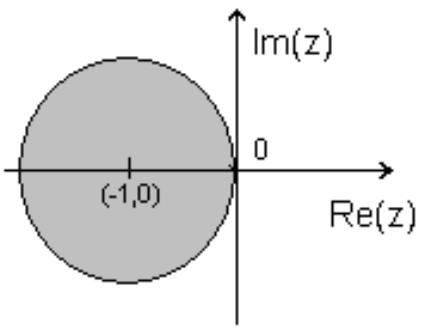


Рисунок 1.1. Область устойчивости метода Эйлера

Говорят, что метод называется A-устойчивым [42], если область его абсолютной устойчивости включает в себя полуплоскость Re(z) $<$ 0 (рисунок 1.2, а). A-устойчивые методы могли бы быть использованы для решения жестких задач, поскольку шаг интегрирования при их использовании не ограничен устойчивостью, однако класс таких методов весьма узок. Например, среди явных линейных многошаговых методов нет A-устойчивых. Доказано также, что среди неявных линейных многошаговых методов нет A-устойчивых методов, имеющих порядок точности выше второго (так называемый второй барьер Дальквиста, [2]). Однако для многих задач требование A-устойчивости избыточно и его можно заменить менее строгими требованиями жесткой устойчивости или $A(\alpha)$-устойчивости. Определение жесткой устойчивости следующее: пусть $R1 = z : Re(z) < -a$, $R2 = \{z : -a \leq Re(z) < 0, -c \leq Im(z) \leq c\}$, где a, c – некоторые положительные константы. Тогда говорят, что метод обладает свойством жесткой устойчивости (stiff stability), если его область устойчивости включает в себя $R1 \cup R2$ [46]. Область устойчивости жестко устойчивого метода схематично изображена на рисунке 1.2, в. Численный метод решения задачи Коши называют $A(\alpha)$-устойчивым, если область его абсолютной устойчивости включает угол $|\arg(-z)| < \alpha$. (рисунок 1.2, б). В частности, при $\alpha = \frac{\pi}{2}$ определение $A(\alpha)$-устойчивости совпадает с определением A-устойчивости. Свойство жесткой устойчивости является более сильным по отношению к свойству $A(\alpha)$-устойчивости в том смысле, что если метод обладает свойством жесткой устойчивости, то он обладает и свойством $A(\alpha)$-устойчивости, но не наоборот [46].

Известно, что среди явных линейных многошаговых методов нет $A(\alpha)$-устойчивых ни при каком $\alpha > 0$. Однако среди неявных линейных многошаговых методов имеются $A(\alpha)$-устойчивые методы высокого порядка точности [2].

Рассмотрим устойчивость некоторых широко распространенных методов решения систем ОДУ. Для явных методов Рунге–Кутты

$$K_i = y_n + h \sum_{j=0}^{i-1} a_{ij} f(t_n + c_j h, K_j),$$

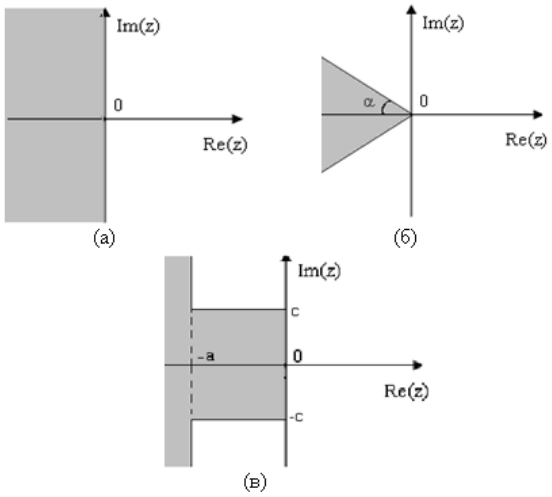


Рисунок 1.2. Области устойчивости A-устойчивого (а), A(α)-устойчивого (б) и жестко-устойчивого (в) методов

$$y_{n+1} = y_n + h\sum_{j=0}^{s-1} b_j f(t_n + c_j h, K_j)$$

решение модельного уравнения на шаге $n+1$ может быть выражено через решение на шаге n следующим образом:

$$y_{n+1} = R(z)y_n,$$

где $R(z)$– полином следующего вида [42]:

$$R(z) = 1 + z\sum_j b_j + z^2\sum_{j,k} b_j a_{jk} + z^2\sum_{j,k,l} b_j a_{jk} a_{kl} + ...$$

Будем говорить, что $R(z)$ – полином устойчивости явного метода Рунге–Кутты. Можно показать [42], что

$$R(z) = 1 + \sum_{k=1}^{p} \frac{z^k}{k!} + O(z^{p+1})$$

и, соответственно, в случае, если порядок точности метода p и количество стадий s совпадают (что возможно при $p \leq 4$), то

$$R(z) = 1 + \sum_{k=1}^{s} \frac{z^k}{k!}.$$

Возвращаясь к проведенному выше анализу явного метода Эйлера (который является явным методом Рунге–Кутты 2-го порядка), мы видим, что для него $R(z) = 1 + z$. Итак, введя определение полинома устойчивости, мы можем определить область устойчивости для явных методов Рунге–Кутты как множество точек z, таких что $|R(z)| \leq 1$. Введем также определение интервала устойчивости, под которым будем понимать подмножество точек действительной оси, для которых $|R(z)| \leq 1$ [42]. Для методов порядков 1-4 области устойчивости приведены на рисунке 1.3 [29]. Из рисунка видно, что методы Рунге–Кутты не являются устойчивыми ни по одному из приведенных выше типов, и область их устойчивости достаточно невелика. Вследствие этого, они плохо подходят для решения жестких систем. Впрочем, далее будет показано, что, манипулируя видом полинома R, можно сконструировать метод со значительно более широким интервалом устойчивости (правда, за счет потери точности), что все же делает возможным (и в некоторых случаях очень эффективным) применение явных методов Рунге–Кутты к умеренно жестким задачам.

Другая часто используемая группа методов – линейные многошаговые методы, в частности методы Адамса [2, 63]. Рассмотрим явный метод Адамса (или, как его еще называют, метод Адамса–Башфорта):

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} \beta_j f(t_{n-j}, y_{n-j}).$$

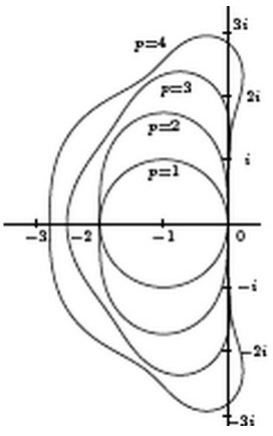


Рисунок 1.3. Области устойчивости методов Рунге–Кутты

Применяя этот метод к модельному уравнению, получим

$$y_{n+1} - (1 + z\beta_0)y_n - \sum_{j=1}^{k-1} z\beta_j y_{n-j} = 0.$$

Подставляя в это выражение $\zeta^i = y_i$ и поделив на ζ^{n-l+1}, мы получим характеристический полином

$$Y(\zeta) = \zeta^k - (1 - z\beta_0)\zeta^{k-1} - \sum_{j=2}^{k-1} z\beta_j \zeta^{k-j}.$$

Можно показать, что верно следующее утверждение: метод Адамса является устойчивым тогда и только тогда, когда корни характеристического полинома по модулю не превышают 1, при этом кратные корни по модулю строго меньше 1 [42]. Исходя из этого, области устойчивости явных методов Адамса при $k = 2..5$ имеют вид, представленный на рис. 1.4 (при $k = 1$ метод Адамса сводится к методу Эйлера).

Как видно из рисунка, все эти методы имеют крайне ограниченную область устойчивости. Для неявных методов Адамса аналогичным образом получаем области устойчивости, изображенные на рисунке 1.5. Метод с k = 1 представляет собой метод трапеций и является A-устойчивым, методы с $k = 2...5$ имеют ограниченный интервал устойчивости, хотя и значительно больший, чем у явных методов.

Аналогичным образом (но с привлечением достаточно громоздких вычислений) можно построить области устойчивости для методов типа предиктор-корректор, использующих в качестве предиктора явный метод Адамса, а в качестве корректора – неявный. Такого рода методы также имеют ограниченную область устойчивости, при этом длина интервала устойчивости лежит между длинами интервалов устойчивости явных и неявных методов, то есть, говоря другими словами, использование схемы типа предиктор-корректор увеличивает устойчивость явных методов Адамса [42]. Теперь рассмотрим неявный метод Эйлера:

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

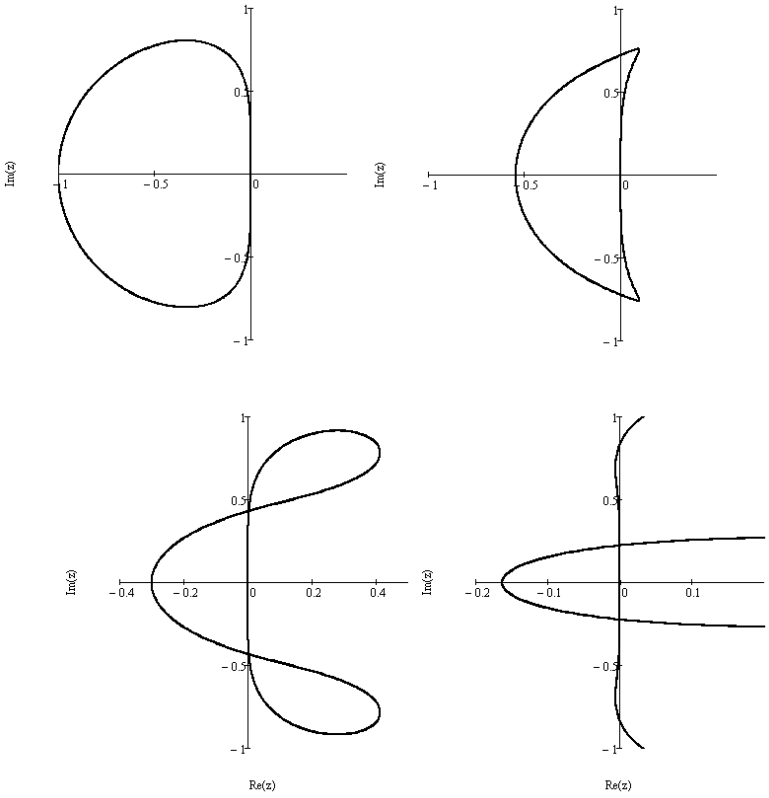


Рисунок 1.4. Области устойчивости явных методов Адамса порядков 2–5

Полином устойчивости для данного метода имеет вид $R(z) = \frac{1}{1-z}$, и поэтому область устойчивости включает в себя всю комплексную плоскость за исключением круга радиуса 1 с центром в точке (1,0). Таким образом, неявный метод Эйлера является A-устойчивым. Для правила трапеций область устойчивости совпадает с отрицательной полуплоскостью, соответственно, метод является A-устойчивым. Функция устойчивости для метода трапеций имеет вид $R(z) = \frac{1+\frac{z}{2}}{1-\frac{z}{2}}$. Если z имеет достаточно большую по модулю действительную часть, то очевидно, что $|R(z)|$ близко к единице, в результате чего метод может, оставаясь формально стабильным, то есть сохраняя ошибку ограниченной, недостаточно компенсировать жесткость системы. В таком случае полученное численное решение будет колебаться вокруг точного, иногда с довольно значительной амплитудой. С другой стороны, возвращаясь к неявному методу Эйлера, мы видим, что $\lim_{z\to\infty} R(z) = 0$, и поэтому при больших по модулю z устойчивость остается высокой. Вслед за Б. Эле [34] введем соответствующее определение: Метод называется L-устойчивым, если $\lim_{z\to\infty} R(z) = 0$. Неявный метод Эйлера является L-устойчивым, метод трапеций – нет. Итак, как мы видим, из широко распространенных методов

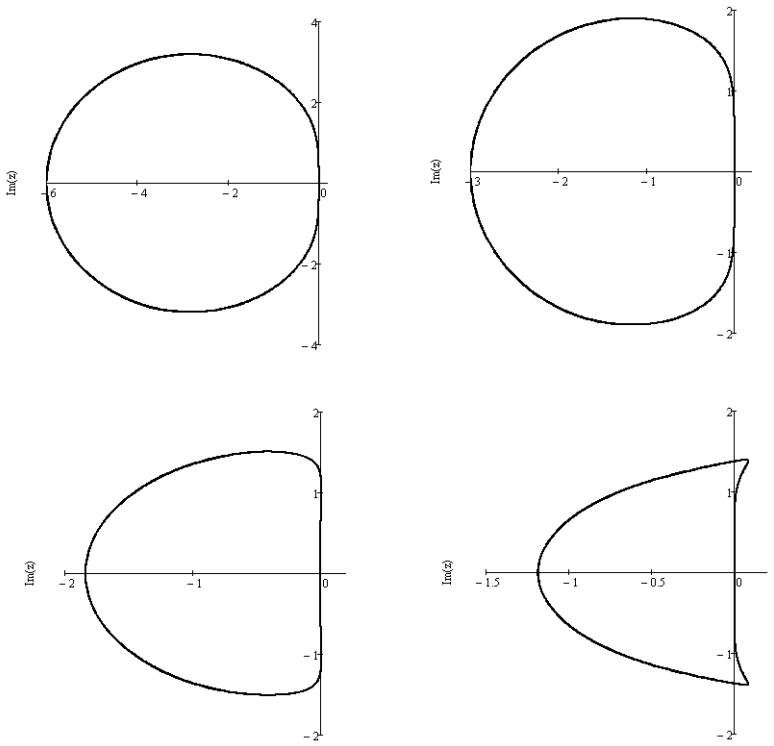


Рисунок 1.5. Области устойчивости неявных методов Адамса

достаточной устойчивостью для решения жестких задач обладают только несколько методов: неявный метод Эйлера, метод трапеций (он же неявный метод Адамса с k = 1) и в определенной степени неявные методы Адамса с k > 1. Свойством L-устойчивости среди них обладает только неявный метод Эйлера. Однако существует множество других методов, которые имеют различные преимущества при решении жестких систем. Нужно отметить, что теоретические свойства A-устойчивости, L-устойчивости и т.д. являются лишь достаточным, но не необходимым условием для того, чтобы та или иная система могла быть эффективно решена. Задача заключается в том, чтобы подобрать (или построить) максимально эффективный метод, обладающий достаточной для решения данной задачи устойчивостью, причем под эффективностью в данном случае подразумевается как вычислительная эффективность, так и другие параметры, определяющие качество решения, например, точность, масштабируемость и т.д.

1.1.6. Неявные методы Рунге–Кутты

Напомним, что метод Рунге–Кутты в общем виде может быть представлен в следующей форме:

$$K_i = y_n + h\sum_{j=0}^{i-1} a_{ij} f(t_n + c_j h, K_j),$$

$$y_{n+1} = y_n + h\sum_{j=0}^{s-1} b_j f(t_n + c_j h, K_j).$$

Коэффициенты a,b,c удобно представлять в виде таблицы Бутчера [28]:

c_0	a_{00}	a_{01}	...	
c_1	a_{10}			
...	⋮			
...	...			
c_s				
	b_0	b_1	...	b_s

Если $a_{ij} = 0$ для всех i ≤ j, то метод будет явным, иначе – неявным. Выбор коэффициентов a, b, c оказывает определяющее влияние на характеристики метода. Существует множество способов выбора этих коэффициентов таким образом, чтобы добиться желаемой точности и устойчивости. В частности, множество методов основано на следующей теореме [42].

Теорема (Бутчер). Если коэффициенты a,b,c метода Рунге–Кутты удовлетворяют следующим условиям:

$$B(p): \quad \sum_{i-1}^{s} b_i c_i^{q-1} = \frac{1}{q}, \quad q = 1, ..., p,$$

$$C(\eta): \quad \sum_{j=1}^{s} a_{ij} c_j^{q-1} = \frac{c_i^q}{q}, \quad i = 1, ..., s, \ q = 1, ..., \eta,$$

$$D(\zeta): \quad \sum_{j=1}^{s} a_{ij} b_i c_i^{q-1} = \frac{b_j}{q(1 - c_j^q)}, \quad j = 1, ..., s, \ q = 1, ..., \zeta,$$

то при $p \le \eta + \zeta + 1$ и $p \le 2\eta + 2$ этот метод имеет порядок точности p.

Далее рассмотрим некоторые неявные методы Рунге–Кутты, используемые для решения жестких задач.

Методы Гаусса. Методы Гаусса, также известные как методы Кунцмана–Бутчера [42], основаны на квадратурных формулах Гаусса, соответственно коэффициенты c для этих методов являются корнями ортогональных полиномов Лежандра:

$$L_s(x) = (s!)^{-1}\frac{d^s}{dx^s}(x^2 - x)^s.$$

Методы Гаусса имеют порядок точности $2s$, где s – количество стадий метода и являются A-устойчивыми. Метод 4-го порядка точности [42]:

$$\begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} - \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Метод 5-го порядка точности [42]:

$$\begin{array}{c|ccc} 1/2\text{-}\sqrt{15}/10 & 5/36 & 2/9\text{-}\sqrt{15}/15 & 5/36\text{-}\sqrt{15}/30 \\ 1/2 & 5/36\text{+}\sqrt{15}/24 & 2/9 & 5/36\text{-}\sqrt{15}/24 \\ 1/2\text{+}\sqrt{15}/10 & 5/36\text{+}\sqrt{15}/30 & 2/9\text{+}\sqrt{15}/15 & 5/36 \\ \hline & 5/18 & 4/9 & 5/18 \end{array}.$$

Методы Радо. Аналогично методам Гаусса, коэффициенты данных методов получены из квадратурных формул особого вида, в данном случае – формул Радо [42]. Значения c вычисляются как корни полиномов вида

$$L_s(x) = \frac{d^{s-1}}{dx^{s-1}}(x^s(x-1)^{s-1}),$$

$$L_s(x) = \frac{d^{s-1}}{dx^{s-1}}(x^{s-1}(x-1)^s).$$

Если коэффициенты c получены первым способом, а коэффициенты a – в соответствии с условием D теоремы Бутчера, то такие методы обозначают IA. Если с получены вторым способом, а коэффициенты a – из условия C, то

получаем методы Радо IIA. Обе группы методов являются A-устойчивыми и имеют порядок точности $2s-1$, где s – количество стадий метода. Метод Радо IA 3-го порядка точности [42]:

0	1/4	-1/4
2/3	1/4	5/12
	1/4	3/4

Метод Радо IA 5-го порядка точности [42]:

0	1/9	(-1-$\sqrt{6}$)/18	(-1+$\sqrt{6}$)/18
(6-$\sqrt{6}$)/10	1/9	(88+7$\sqrt{6}$)/360	(88-43$\sqrt{6}$)/360
(6+$\sqrt{6}$)/10	1/9	(88+43$\sqrt{6}$)/360	(88-7$\sqrt{6}$)/360
	1/9	(16+$\sqrt{6}$)/36	(16-$\sqrt{6}$)/36

Метод Радо IIA 3-го порядка точности [42]:

1/3	5/12	-1/12
1	3/4	1/4
	3/4	1/4

Метод Радо IIA 5-го порядка точности [42]:

(4-$\sqrt{6}$)/10	(88-7$\sqrt{6}$)/360	(269-169$\sqrt{6}$)/1800	(-2+3$\sqrt{6}$)/225
(4+$\sqrt{6}$)/10	(269+169$\sqrt{6}$)/1800	(88+7$\sqrt{6}$)/360	(-2-3$\sqrt{6}$)/225
1	(16-$\sqrt{6}$)/36	(16+$\sqrt{6}$)/36	1/9
	(16-$\sqrt{6}$)/36	(16+$\sqrt{6}$)/36	1/9

Методы Лобатто. Для методов Лобатто [42] коэффициенты c вычисляются из квадратурных формул Лобатто:

$$L_s(x) = \frac{d^{s-2}}{sx^{s-2}}(x^{s-1}(x-1)^s).$$

Коэффициенты b находятся из условия $B(2s-2)$, а коэффициенты a либо из условия C (методы Лобатто IIIA), либо из условия D (методы Лобатто IIIB),

либо (методы Лобатто IIIC) из условия C($s-1$) при $a_{i1} = b_1$. Методы Лобатто являются A-устойчивыми и имеют порядок точности $2s-2$. Рассмотрим несколько примеров методов Лобатто.

Метод Лобатто IIIA 4-го порядка [42]:

0	0	0	0
1/2	5/24	1/3	-1/24
1	1/6	2/3	1/6
	1/6	2/3	1/6

.

Метод Лобатто IIIA 6-го порядка [42]:

0	0	0	0	0
(5-$\sqrt{5}$)/10	(11+$\sqrt{5}$)/120	(11+$\sqrt{5}$)/120	(11+$\sqrt{5}$)/120	(-1+$\sqrt{5}$)/120
(5+$\sqrt{5}$)/10	(11-$\sqrt{5}$)/120	(25+13$\sqrt{5}$)/120	(25+$\sqrt{5}$)/120	(-1-$\sqrt{5}$)/120
1	1/12	5/12	5/12	1/12
	1/12	5/12	5/12	1/12

Метод Лобатто IIIB 4-го порядка [42]:

0	1/6	-1/6	0
1/2	1/6	1/3	0
1	1/6	5/6	0
	1/6	2/3	1/6

Метод Лобатто IIIB 6-го порядка [42]:

0	1/12	(-1-$\sqrt{5}$)/24	(-1+$\sqrt{5}$)/24	0
(5-$\sqrt{5}$)/10	1/12	(25+$\sqrt{5}$)/120	(25-13$\sqrt{5}$)/120	0
(5+$\sqrt{5}$)/10	1/12	(25+13$\sqrt{5}$)/120	(25-$\sqrt{5}$)/120	0
1	1/12	(11-$\sqrt{5}$)/24	(11+$\sqrt{5}$)/24	1/12
	1/12	5/12	5/12	1/12

Метод Лобатто IIIC 4-го порядка [42]:

0	1/6	-1/3	1/6
1/2	1/6	5/12	-1/12
1	1/6	2/3	1/6
	1/6	2/3	1/6

Метод Лобатто IIIC 6-го порядка [42]:

0	1/12	-$\sqrt{5}$/12	$\sqrt{5}$/12	-1/12
(5-$\sqrt{5}$)/10	1/12	1/4	(10-7$\sqrt{5}$)/60	$\sqrt{5}$/60
(5+$\sqrt{5}$)/10	1/12	(10+7$\sqrt{5}$)/60	1/4	-$\sqrt{5}$/60
1	1/12	5/12	5/12	1/12
	1/12	5/12	5/12	1/12

1.1.7. Диагонально-неявные методы

При решении нежестких задач неявные методы Рунге–Кутты практически не находят применения из-за высоких вычислительных затрат. Более того, даже для жестких задач вычислительные затраты этих методов часто оказываются неприемлемыми. Действительно, на каждом шаге они требуют решения системы из ns алгебраических уравнений (n – количество дифференциальных уравнений в системе; s – количество стадий). Однако во многих случаях возможно значительно упростить вычислительную схему, потребовав, чтобы матрица коэффициентов a_{ij} была нижнетреугольной. В таком случае на каждом шаге вместо системы из ns уравнений нужно решить s систем из n уравнений, что значительно проще. Если таблица Бутчера неявного метода Рунге–Кутты является нижнетреугольной, то такой метод называется диагонально-неявным методом Рунге–Кутты (DIRK, diagonally-implicit Runge–Kutta) [42]. Вычислительные затраты диагонально-неявного метода можно еще уменьшить, потребовав, чтобы для всех i $a_{ij} = a$. Такие методы называются однократно диагонально-неявными (SDIRK, singly diagonally-implicit Runge–Kutta [42]). В случае SDIRK-метода за счет равенства диагональных элементов возможно при каждом решении системы из n уравнений использовать одну и ту же матрицу Якоби, вычислив ее LU–разложение единственный раз на шаг. Подробное описание способа выбора коэффициентов для диагональных методов Рунге-Кутты можно найти в [42], здесь приведем лишь конечные результаты. Все приведенные нами методы будут относиться к SDIRK-методам.

Метод 3-го порядка точности:

$(3+\sqrt{3})/6$	$(3+\sqrt{3})/6$	0
$(3-\sqrt{3})/6$	$-\sqrt{3}/3$	$(3+\sqrt{3})/6$
	1/2	1/2

Следующие два метода относятся к группе так называемых вложенных методов и представляют собой пары методов Рунге-Кутты разного порядка точности, отличающиеся только коэффициентами b. Вложенные методы позволяют при небольшом увеличении затрат организовать автоматическое управление длиной шага, основанное на оценке погрешности как разности результатов, полученных методами пары. В случаях, когда оценка погрешности не требуется, один из рядов коэффициентов b может быть проигнорирован.

Метод NT1 [66, 54]:

0	0			
5/6	5/12	5/12		
10/21	95/588	-5/49	5/12	
1	59/600	-31/75	539/600	5/12
	59/600	-31/75	539/600	5/12
	-37/600	-31/75	1813/6600	37/132

L-стабильный метод [42]:

1/4	1/4				
3/4	1/2	1/4			
11/20	17/50	-1/25	1/4		
1/2	371/1360	-137/2720	15/544	1/4	
1	25/24	-49/48	125/16	-85/12	1/4
	25/24	-49/48	125/16	-85/12	1/4
	59/48	-17/96	225/32	-85/12	0

1.1.8. Метод Гира

Методом Гира в русскоязычной литературе (например, [6]) называют один из множества методов решения жестких задач Коши, основанных на так называемых формулах дифференцирования назад (backward differentiation

formulae, BDF [63]) q-го порядка:

$$\sum_{i=0}^{q} \alpha_{n,i} y_{n,i} + h\beta_n f(t_n, y_n) = 0.$$

Известно, что формулы порядков 1-6 обладают свойством $A(\alpha)$-устойчивости и жесткой устойчивости [46]. Формулы порядков 1 и 2 являются A-устойчивыми. Коэффициенты α и β для формул приведены в табл. 1.1.

Таблица 1.1. Коэффициенты BDF порядков 1-6 [46]

q	α_6	α_5	α_4	α_3	α_2	α_1	α_0	β
1						1	-1	1
2					1	-4/3	1/3	2/3
3				1	-18/11	9/11	-2/11	6/11
4			1	-48/25	36/25	-16/25	3/25	12/25
5		1	-300/137	300/137	-200/137	75/137	-12/137	60/137
6	1	-360/147	-72/147	-450/147	-400/147	-225/147	-10/147	60/147

Благодаря относительной простоте и высокой устойчивости алгоритм приобрел большую популярность и считается «промышленным стандартом» для решения систем большой жесткости. Существуют программные реализации метода, использующие крайне сложные схемы контроля погрешности, на основе которой автоматически выбирается порядок и шаг метода. Широко используемой является реализация из библиотеки SUNDIALS (а конкретнее, CVODE, основанной в свою очередь на VODE), разработанная в Ливерморской национальной лаборатории США [44].

Данная реализация основана на представлении решения в виде вектора Нордсика (Nordsieck vector). Нордсик предложил рассматривать решение дифференциального уравнения $y' = f(x, y)$ как нахождение аппроксимирующего полинома, представленного вектором, содержащим q производных y:

$$Z = [y_n, hy_n^{'}, ..., \frac{h^q y_n^{(q)}}{q!}].$$

Однако итерационные формулы, полученные непосредственно из такого представления полинома, оказались нестабильными (подробнее см. [63]). Для решения этой проблемы Нордсик предложил ввести q коэффициентов $l_0 \dots l_{q-1}$, что позволило (при правильном выборе этих коэффициентов) до-

биться стабильности. Можно показать, что после такой модификации метод Нордсика эквивалентен некоторому неявному многошаговому методу [63].

Легко заметить, что методы, основанные на BDF, являются неявными. Соответственно, для получения значения функции на каждом шаге необходимо решать систему уравнений:

$$y_n - h\beta_n f(t_n, y_n) - \sum_{i>0} \alpha_{n,i} y_{n-i} = 0.$$

В общем случае эта система нелинейна и для ее решения обычно используют метод Ньютона. Напомним, что метод Ньютона для решения системы нелинейных уравнений $f(x) = 0$ представляет собой итерационную процедуру следующего вида:

$$x_n - x_{n-1} - J_{n-1}^{-1} f(x_{n-1}),$$

где J_{n-1} – матрица Якоби системы в точке x_{n-1}, n – номер шага.

Однако вести вычисления по вышеприведенной формуле неудобно, поскольку она требует трудоемкой операции обращения матрицы. Поэтому обычно вместо этого решают одним из численных методов систему линейных уравнений относительно $\triangle x_n$:

$$J_{n-1} \triangle x_n = -f(x_{n-1}),$$

где $\triangle x_n = x_n - x_{n-1}$. Затем полагают $x_n = x_{n-1} + \triangle x_n$. Для задач относительно небольшой размерности решение линейной системы лучше всего осуществлять при помощи LU–разложения, однако больших и, особенно, сильно разреженных систем гораздо более выгодным является использование одного из итерационных методов, основанных на теории подпространств Крылова (Krylov subspace methods). При использовании надлежащего предобусловливания данные методы на настоящее время являются наиболее эффективными для решения разреженных СЛАУ, а также в некоторых случаях оказываются эффективнее прямых методов и для плотных систем. В частности, упомянутая библиотека SUNDIALS содержит целую группу таких методов. Среди преимуществ итерационных методов можно также выделить относительную простоту и эффективность параллельной реализации. Подробное описание

итерационных методов, можно найти в работе [22], исследование эффективности методов предобусловливания – в работах [23] и [24].

1.1.9. Явно-неявные методы

Во многих случаях решаемая система уравнений может быть представлена в виде

$$\frac{dx}{dt} = f(x,t) + g(x,t)$$

таким образом, что $\frac{dx}{dt} = f(x,t)$ – нежесткая система и может решаться явным методом, а $\frac{dx}{dt} = g(x,t)$ – жесткая система и, соответственно, должна решаться неявным методом. Тогда, как правило, для системы в целом эффективно применение одного из явно-неявных методов (IMEX, от IMplicit-EXplicit) [61, 21]. Известно множество явно-неявных схем, составленных из различных пар методов. Мы рассмотрим вариант, составленный из явного и неявного методов Рунге–Кутты (в литературе такие методы обычно называются IMEX-RK). Пусть имеется два метода Рунге–Кутты: явный (с коэффициентами $\tilde{a}_{ij}$, $\tilde{b}_i$ и $\tilde{c}_i$) и диагонально-неявный (с коэффициентами $a_{i,j}$, b_i и c_i). Тогда

$$y_{n+1} = y_n + h\sum_{i=1}^{s} \tilde{b}_i f(t_n + h\tilde{c}_i, U_i) + h\sum_{i=1}^{s} b_i g(t_n + c_i, U_i).$$

Промежуточные значения U_i вычисляются следующим образом:

$$U_i = y_n + h\sum_{j=1}^{i-1} \tilde{a}_{i,j} f(t_n + h\tilde{c}_j, U_j) + h\sum_{j=1}^{i} a_{i,j} g(t_n + hc_j, U_j).$$

Из этой формулы можно выделить явную часть (т. е. слагаемые, которые могут быть вычислены непосредственно без решения системы уравнений):

$$\Xi_i = y_n + h\sum_{j=1}^{i-1} \tilde{a}_{i,j} f(t_n + h\tilde{c}_j, U_j) + h\sum_{j=1}^{i-1} a_{i,j} g(t_n + hc_j, U_j).$$

Тогда система уравнений, которая должна быть решена на каждом шаге, будет выглядеть следующим образом:

$$U_i = \Xi_i + ha_{ii}g(t_n + hc_i, U_i).$$

Можно видеть, что данная система гораздо проще для решения, чем система, получающаяся при решении полностью неявным методом, поэтому эффективность явно-неявной схемы может быть значительно выше. Приведем коэффициенты некоторых распространенных реализаций явно-неявных методов Рунге–Кутты [50]:

0	0			
c_2	γ	γ		
c_3	$-4{,}30002662176923$	$2{,}26541338346372$	γ	
1	b_1	0	b_3	γ
	b_1	0	b_3	γ

0	0			
c_2	$0{,}871733043016917$	0		
c_3	$-3{,}06478674186224$	$1{,}466040025065192$	0	
1	$0{,}21444560762133$	$0{,}71075364965269$	$0{,}07480074272597$	0
	b_1	0	b_3	γ

$c_2 = 0{,}87173304301691$, $c_3 = -1{,}59874671679705$, $\gamma = 0{,}43586652150845$, $b_1 = 0,60424832458800$, $b_3 = 0,04011484609646$.

Отметим также метод четвертого порядка точности ARK5 [50], который является одним из наиболее распространенных на практике методов из данной группы и имеет 8 стадий.

1.1.10. Методы Розенброка

Рассмотрим некоторый диагонально-неявный метод Рунге–Кутты. Этот метод на каждом шаге требует решения системы алгебраических уравнений, которое, как правило, производится одним из вариантов метода Ньютона. Если мы ограничимся одной итерацией метода Ньютона, то результирующий метод будет линейно-неявным, т. е. будет требовать на каждом шаге только

решения линейной системы алгебраических уравнений. Общий вид таких методов для автономных систем (т. е. систем, правая часть которых не зависит от времени) следующий:

$$k_i = hf(y_n + \sum_{j-1}^{i-1} \alpha_{i,j} k_j) + hJ \sum_{j=1}^{i} \gamma_{i,j} k_j,$$

$$y_{n+1} = y_n + \sum_{j-1}^{s} b_j k_j,$$

где J – матрица Якоби, вычисленная в точке y_n; s – количество стадий метода. Если мы положим $\gamma_{i,j} = \gamma \forall i$, то вычисления по такой схеме потребуют всего одного решения линейной системы на каждом шаге. Для неавтономной системы придется также учесть зависимость от времени:

$$k_i = hf(t_n + h\sum_{j=1}^{i-1} \alpha_{i,i}, y_n + \sum_{j-1}^{i-1} \alpha_{i,j} k_j) + \frac{\partial f}{\partial y} h \sum_{j=1}^{i} \gamma_{i,j} k_j + \frac{\partial f}{\partial t} h^2 \sum_{j=1}^{i} \gamma_{i,j}.$$

Данное выражение сохраняет свойство диагональной неявности: на каждом шаге требуется решить систему линейных алгебраических уравнений

$$(I - h\gamma_{ii} J)k = 0, k = (k_1, ..., k_s)^T$$

в автономном случае и соответственно

$$(I - (\frac{\partial f}{\partial y} h\gamma_{ii} + \frac{\partial f}{\partial t} h^2 \gamma_{ii}))k = (I - h\gamma_{ii}(\frac{\partial f}{\partial y} - \frac{\partial f}{\partial t} h))k = 0$$

в неавтономном [42].

1.1.11. Явные методы с расширенной областью устойчивости

Во многих практических задачах, в частности при дискретизации систем дифференциальных уравнений в частных производных, возникают системы уравнений «умеренной» жесткости (которые, впрочем, не могут быть эффективно решены обычными явными методами) с относительно невысокими требованиями к точности решения. В таких ситуациях вместо трудоемких неявных методов можно воспользоваться явными методами, специально скон-

струированными таким образом, чтобы иметь достаточно большой интервал устойчивости. Рассмотрим метод Рунге–Кутты:

$$K_i = y_n + h\sum_{j-1}^{s} a_{ij} f(t_n + c_j h, K_j),$$

$$y_{n+1} = y_n + h\sum_{j=1}^{s} b_j f(t_n + c_j h, K_j).$$

Из приведенных формул следует, что существует достаточно большая свобода в выборе коэффициентов a, b, c. Традиционно эти коэффициенты выбираются таким образом, чтобы достичь максимального порядка точности при минимальном значении s. Однако можно подобрать их и таким образом, чтобы получить более широкий интервал устойчивости. Действительно, применяя данный метод к уравнению

$$y' = \lambda y, \lambda \in C, Re(\lambda) < 0$$

и принимая, как и ранее, обозначение $z = h\lambda$, получаем для явного метода

$$y_{n+1} = R(z)y_n = (1 + zb^T(I - zA)^{-1}e)y_n,$$

где $R(z)$ – полином; A и b – соответственно матрица и вектор, составленные из коэффициентов a_{ij}и b_i; I – единичная матрица; e – единичный вектор. Выбирая вид полинома R, мы можем влиять на форму и размеры области устойчивости метода и одновременно на точность. Поэтому для получения практически полезного метода интегрирования приходится выбирать некое компромиссное решение, обеспечивающее и достаточную устойчивость, и достаточную точность. В работе [18] показано, что для модельных уравнений в некоторых случаях такие методы значительно превосходят неявные. На практике наибольшее распространение получили три алгоритма: RKC [59], DUMKA [51] и ROCK [20]. Рассмотрим подробнее метод RKC (Рунге-Кутты-Чебышёва). Данный метод имеет второй порядок точности (существует вариант метода первого порядка точности, однако на практике обычно не применяется). Метод RKC был изначально разработан для решения систем уравнений, возникающих при дискретизации уравнений в частных производных

параболического типа. Этим объясняется относительно невысокий порядок точности, а также выбор полинома R в виде

$$R_j(z) = a_j + b_j T_j(w_0 + w_1 z),$$

где T_j – полиномы Чебышёва первого рода, определяемые следующим рекурсивным соотношением:

$$T_0(x) = 1,$$

$$T_1(x) = x,$$

$$T_j(x) = 2xT_{j-1}(x)T_{j-2}(x),$$

либо в явном виде: $T_j(x) = cos(j \times arccos(x))$. Итерации по данному методу выполняются следующим образом:

$$W_0 = y_n,$$

$$W_1 = W_0 + \tilde{\mu_1} h f(t_n + hc_0, W_0),$$

$$W_j = (1-\mu_j-\nu_j)W_0+\mu_1 W_{j-1}+\nu_j W_{j-2}+\tilde{\mu_j} h f(t_n+hc_{j-1}, W_{j-1})+\gamma_j h f(t_n+hc_0, W_0),$$

$$y_{n+1} = W_s,$$

где

$$\tilde{\mu_1} = b_1 w_1, \tilde{\mu_j} = \frac{2b_j w_1}{b_{j-1}};$$

$$c_0 = 0, c_1 = \frac{c_2}{4w_0}, c_j = \frac{j^2-1}{s^2-1};$$

$$\mu_j = \frac{2b_j w_0}{b_{j-1}};$$

$$\nu_j = \frac{-b_j}{b_{j-2}};$$

$$\gamma_j = -a_{j-1}\tilde{\mu};$$

$$w_0 = 1 + \frac{\epsilon}{s^2}, w_1 = \frac{T_s'(w_0)}{T_s''(w_0)};$$

$$b_j = \frac{T_j''(w_0)}{(T_j'(w_0))^2}, b_0 = b_1 = b_2;$$

$$a_j = 1 - b_j T_j(w_0);$$

$$j = 2, ..., s.$$

Как можно видеть, приведенные формулы зависят от двух внешних параметров: $s \geq 3$ и $\epsilon \geq 0$, ϵ – параметр затухания, цель которого – обеспечение достаточно малого значения R_j на всем интервале устойчивости. Значение ϵ может выбираться в достаточной степени произвольно, однако стоит иметь в виду, что с увеличением ϵ интервал стабильности сокращается. В работе [61] приводится удобное для практических целей выражение для приближенного нахождения длины интервала устойчивости:

$$\beta = \frac{2}{3}(s^2 - 1)(1 - \frac{2}{15}\epsilon).$$

В той же работе рекомендуется значение ϵ=2/13.

1.1.12. Методы с автоматическим обнаружением жесткости

Несмотря на то что в литературе часто принято упрощенное разделение систем уравнений на «жесткие» и «нежесткие» (причем, граница между ними часто проводится весьма субъективно), внимательный анализ определения жесткости, приведенный в начале главы 1 данной работы, говорит о том, что жесткость является локальной характеристикой системы, то есть при различных значениях независимой переменной система может иметь различную жесткость. Очевидно, что для систем уравнений, жесткость которых значительно меняется, нет смысла использовать специализированные устойчивые методы на всем диапазоне изменения независимой переменной. В таких случаях используют методы, периодически анализирующие жесткость системы и выбирающие на основании этого анализа «жесткий» или «нежесткий» алгоритм. Анализ жесткости может производиться разными методами, перечислим некоторые из них [58]:

- непосредственный анализ собственных чисел матрицы Якоби;
- метод итераций Арнольди [35] (находит приближенные значения собственных чисел);

– метод константы Липшица [56];

– анализ ошибки вычислений или необходимого размера шага [58].

Интересен также подход, сочетающий в себе преимущества методов с расширенным интервалом устойчивости и методов с автоматическим обнаружением жесткости. В работах [9, 8] предлагается следующий подход. Рассмотрим метод Рунге–Кутты порядка s с одной дополнительной стадией:

$$K_s = K_{s-1} + h\alpha(f_{s-1} - f_{s-2}),$$

где α – некоторая константа, например $\alpha = 0,001$; $f_i = f(t_n + c_i h, K_i)$. В случае линейной системы уравнений вида $y^{'} = Ay$ значение на следующем шаге определяется полиномом устойчивости $R(hA)$: $y_{(n+1)} = R(hA)y_n$. Для нелинейных систем применим обычный прием – линеаризуем систему в точке t_n. Тогда $y_{(n+1)} \approx R(hJ)y_n$, где J – матрица Якоби. Тот факт, что R представляет собой полином, позволяет нам воспользоваться методом степеней для нахождения приближенного значения собственных чисел матрицы hJ:

$$a = \alpha(f_{s-1} - f_{s-2}), b = f_s - f_{s-1}, \zeta = \frac{b}{a}.$$

Имея ζ в качестве оценки наибольшего по модулю собственного значения матрицы Якоби (и, соответственно, жесткости системы) , значение функции на $n+1-$ м шаге можно будем вычислять при помощи выражения

$$y_{n+1} = K_{s-1} hc(f_{s-1} - f_{s-2}),$$

где c – параметр, значение которого настраивается в зависимости от значения ζ.

При использовании алгоритма автоматического определения жесткости важным является выбор эффективной стратегии вызова этого алгоритма. Наиболее очевидным решением представляется определение жесткости на каждом шаге алгоритма, что может быть невозможно из-за недопустимого увеличения вычислительных затрат. Эту проблему можно решить используя более быстрые (но менее надежные) алгоритмы определения жесткости, например итерации Арнольди, вместо вычисления собственных чисел методом QR-разложения, либо уменьшив количество оценок жесткости. Послед-

него можно добиться, оценивая жесткость на каждом n-м шаге, где n, как правило, подбирается экспериментально. Существует также набор эвристик, позволяющих более качественно определить момент для оценки жесткости системы. Например, в [58] предлагается переоценивать жесткость системы в следующих случаях:

- каждые n шагов;
- если алгоритм изменения шага по времени в течение последних 50 шагов уменьшал шаг не менее 10 раз и

$$h \in [\frac{\bar{h}}{a}, a\bar{h}],$$

где $\bar{h}$ - среднее значение h за некоторый промежуток; a - некоторая константа, например $a = 5$.

1.1.13. Проекционные методы

Еще один подход к решению жестких систем явными методами состоит в использовании так называемых проекционных методов [38]. Идея состоит в следующем: для жестких систем спектр, как правило, содержит набор жестких (с большой по модулю отрицательной действительной частью) и нежестких компонент. В случае, если эти компоненты расположены в нескольких изолированных друг от друга кластерах, мы можем попробовать подавить действие жестких компонент на решение и таким образом обеспечить возможность использования шага про времени большего размера. Сделать это можно, выполнив несколько шагов интегрирования заведомо устойчивым методом. То есть, алгоритм интегрирования будет следующим (см. также рисунок 1.6):

1. Сначала сделать k шагов некоторым заведомо устойчивым внутренним методом. Устойчивость можно обеспечить, например, выбором очень малого шага по времени.

2. Используя полученную внутренним методом аппроксимацию, экстраполировать значение функции на следующем шаге. Эта процедура будет являться внешним методом интегрирования.

Выбор внутреннего метода существенной роли не играет. Важно, чтобы он обеспечивал порядок точности не ниже первого и устойчивость при вы-

бранном значении внутреннего шага. Учитывая названные требования, разумно выбрать метод Эйлера, как требующий минимально возможное количество вычислений правой части системы и поэтому наиболее вычислительно эффективный. Выбор же внешнего метода во многом определяет общую точность и устойчивость метода. Простейший вариант – использование линейной интерполяции для двух последних точек, полученных внутренним методом, что аналогично использованию метода Эйлера в качестве внешнего:

$$y_{\mathrm{N}} = y_{n+k} + \frac{y_{n+k} - y_{n+k-1}}{h_{int}} h_{ext},$$

где y_N – результирующее значение y на шаге; k – количество внутренних шагов; h_{int}, h_{ext}– соответственно внутренний и внешний шаги по времени; ; h_M- общий шаг проекционного метода. Для программной реализации с использованием функций BLAS более удобно вычислять y_N следующим образом:

$$\psi = \frac{h_{\mathrm{M}}}{h_{int}} - k = \frac{h_{ext}}{h_{int}},$$

$$y_N = (\psi + 1) y_{n+k} - \psi y_{n+k-1}.$$

Описанный выше метод известен как проекционный метод Эйлера (projective forward Euler method). Можно показать, что он имеет первый порядок точности [38].

Вычислим, какое преимущество в производительности способен дать проекционный метод Эйлера по сравнению с обычным явным методом Эйлера. Предположим, что время выполнения внешнего интегрирования пренебрежимо мало по сравнению со временем вычисления правой части системы,

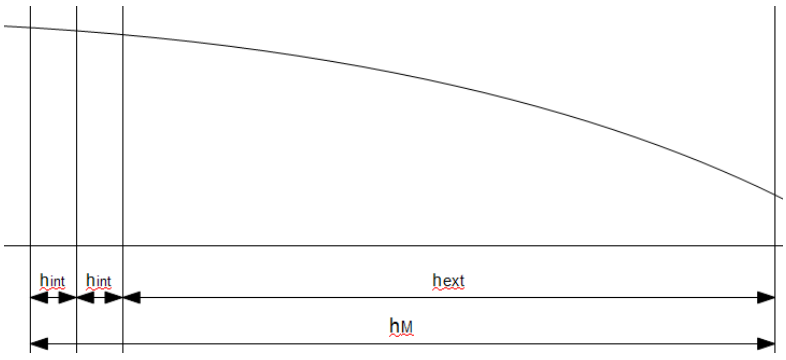


Рисунок 1.6. Проекционный метод.

что разумно, так как аппроксимация требует всего нескольких векторных операций, тогда как вычисление правой части может иметь произвольную сложность. Тогда, если шаг внутреннего интегратора выбран максимально возможным для данной системы, проекционный метод за счет большего значения шага внешнего интегратора будет быстрее в $E = \frac{h_{\text{M}}}{kh_{int}} = 1 + \frac{h_{ext}}{kh_{int}}$ раз.

Более высокой точности можно добиться, выбрав другой внешний интегратор. Например, мы можем увеличить количество шагов внутреннего интегратора и выполнить экстраполяцию более высокого порядка. Так, в [47] приводятся варианты проекционных методов Рунге-Кутты и Адамса-Башфорта второго порядка точности. Эти методы основаны на следующем принципе: выполняется два этапа итераций внутреннего метода и вычисляются соответственно две аппроксимации производной, взвешенная сумма которых используется во внешнем интеграторе. Пусть $\alpha = [0, 1]-$некоторая константа, тогда

$$y^{'} = \alpha y_1^{'} + (1 - \alpha) y_2^{'}.$$

Поскольку неявные методы в целом более устойчивы, чем явные, можно расширить область устойчивости проекционного метода, используя в качестве внешнего некоторый неявный метод, но это сведет на нет преимущество результирующего метода в производительности перед известными жесткими неявными методами, например методом Гира и неявными методами Рунге-Кутты. Однако, учитывая наличие сглаживающих шагов, выполняемых внутренним интегратором, можно упростить неявную часть метода, заменив решение системы уравнений методом Ньютона на простую итерацию. Алгоритм будет тогда иметь следующий вид:

1. Выполнить k шагов внутренним интегратором.
2. Вычислить y_N.
3. Выполнить k шагов внутренним интегратором от полученного y_{N}.
4. Скорректировать y_{N} по формуле

$$y_N = y_{n+k} + \alpha M(y_{n+k} - y_{n+k-1}) + (1 - \alpha) M(y_{N+k} - y_{N-k-1}).$$

5. Повторить шаги 3,4 до достижения сходимости.

В работе [38] показано, что такой метод имеет первый порядок точности, за исключением значения $\alpha=\frac{M+2k-1}{2(M+k)}$, при котором метод имеет второй порядок точности.

В случае сложной конфигурации спектра системы часто используют так называемые телепроекционные методы [39], которые используют некоторый проекционный метод в качестве внутреннего интегратора другого проекционного метода (этот результирующий метод и носит название телепроекционного). Для некоторых систем такая техника позволяет еще увеличить шаг интегрирования. Телепроекционный метод может быть представлен следующей рекурсивной процедурой (L - уровень вложенности телепроекционного метода):

1. Выполнить внутреннее интегрирование.
   1.1. Если L>0, выполнить k шагов интегрирования проекционным методом с L=L-1.
   1.2. Иначе выполнить k шагов интегрирования методом Эйлера (или другим простым внутренним методом).
2. Выполнить шаг внешнего интегрирования.

На практике обычно достаточно использовать L=2, хотя для отдельных систем могут понадобиться и более высокие значения.

Метод с автоматическим выбором размера шага Для построения методов с автоматическим выбором шага используется множество различных методов. Наиболее общим является использование правила Рунге [3]:

1. Вычислить решение y_h с шагом h.
2. Вычислить решение $y_{h/2}$ с шагом $h/2$.
3. Имея y_hи $y_{h/2}$, вычислить оценку погрешности ϵ. Для этого обычно используют вариант экстраполяции Ричардсона:

$$\epsilon = \frac{y_{h/2} - y_h}{2^p - 1},$$

   где p - порядок точности метода.

Очевидным недостатком такого решения является необходимость вычислять решение 2 раза на каждом шаге. По этой причине были разработаны различные приемы, позволяющие получить оценку погрешности с меньшими вычис-

лительными затратами. Для методов Рунге-Кутты наиболее распространенным является использование вложенных методов [63] - пары методов, отличающихся только набором коэффициентов b, но при этом имеющих разный порядок точности. В этом случае мы можем, используя всего одно дополнительное вычисление правой части, получить оценку погрешности на шаге. Очевидно, что вложенные методы Рунге-Кутты с количеством стадий больше 1 имеют преимущество в производительности перед методами, использующими правило Рунге. По этой причине они используются практически во всех современных математических пакетах, например, системы MATLAB и GNU Octave используют метод Дормана-Принса, имеющий порядок точности 5(4).

Для проекционных методов в работе [47] используется вариант правила Рунге, однако более эффективно будет сконструировать вложенный проекционный метод. Наиболее очевидный вариант - использовать два внешних интегратора, обеспечивающих соответственно первый и второй порядок точности. Напомним, что внешний интегратор, как правило, не требует вычисления правой части системы, однако внешний интегратор второго порядка точности может потребовать большее количество шагов внутреннего интегратора, чем метод первого порядка точности.

Таким образом, алгоритм проекционного метода с автоматическим вычислением погрешности будет следующим:

1. Выполнить s_1 шагов внутренним интегратором. Значение s_1 выберем таким образом, чтобы s_1шагов внутреннего интегратора обеспечивали решение системы первого порядка точности.
2. Выполнить s_2 шагов внутренним интегратором. Значение s_2 выберем таким образом, чтобы $s_1 + s_2$ шагов внутреннего интегратора обеспечивали решение системы второго порядка точности.
3. Произвести внутренне интегрирование дважды и получить два решения y^1 и y^2 первого и второго порядка точности соответственно.
4. Вычислить ошибку $\epsilon = ||y^2 - y^1||$.
5. Если $\epsilon > \epsilon_N$, где ϵ_N - некоторая константа, пересчитать решение с меньшим шагом интегрирования. Важно отметить, что при удачно выбранной длине шага внутреннего интегрирования, обеспечивающей достаточное демпфирование жестких компонент спектра, повторять внутреннее интегрирование не нужно.

Гибридные методы Как уже говорилось, жесткость - локальная характеристика системы, т. е. в общем случае жесткость системы может изменяться на каждом шаге интегрирования. Действительно, классическое определение использует в качестве численной характеристики жесткости значения собственных чисел линеаризованной матрицы правой части системы при каком-то определенном значении независимой переменной. С другой стороны, нежесткие методы обычно имеют преимущество перед жесткими в точности или производительности. Используя либо технику оценки собственных значений, что соответствует теоретическому определению жесткости, либо используя технику контроля размера шага (в соответствии с практическим определением жесткости: система является жесткой, если при решении нежесткими методами она требует слишком малого размера шага), можно построить множество комбинированных алгоритмов, автоматически выбирающих метод решения в зависимости от локальных характеристик системы. Например, алгоритм может быть следующим:

1. Вычисляем решение некоторым явным методом Рунге-Кутты:

$$K_i = y_n + h \sum_{j=0}^{i-1} a_{ij} f(t_n + c_j h, K_j)$$

$$y_{n+1} = y_n + h \sum_{j=0}^{s-1} b_j f(t_n + c_j h, K_j),$$

   например методом Богацки-Шампайна.
2. Вычисляем оценку главного собственного значения следующим образом:

$$\lambda = \frac{K_{s-1} - K_{s-2}}{K_{s-2} - K_{s-3}}.$$

3. Если $\lambda < \lambda_T$, завершить шаг методом Рунге-Кутты.
4. Иначе, перевычислить шаг жестким методом. В некоторых случаях, например если мы возьмем методы Богацки-Шампайна и проекционный метод Эйлера, мы можем реиспользовать уже вычисленное значение правой части в точке начала шага, сокращая таким образом вычислительные расходы. Более того, вычисления правой части функции разными методами независимы друг от друга и могут выполняться па-

раллельно, что позволяет еще сократить общее время вычислений, если количество процессоров достаточно велико.

Для пары методов Богацки-Шампайна - PFE, на шаге будет требоваться в худшем случае 6 вычислений правой части: 3 вычисления для метода Богацки-Шампайна и 4 для проекционого метода, при том, что одно вычисление уже выполнено. Однако использовать оба метода на каждом шаге нерационально, более эффективно будет в случае, если был обнаружен жесткий участок, следующие n шагов сразу начинать выполнение с проекционного метода. Тогда в худшем случае на каждом шаге в среднем будет $4 + \frac{3}{n}$ вычислений правой части.

1.2. Параллельное решение систем ОДУ

Современные вычислительные устройства поддерживают множество форм параллельного выполнения вычислений (причем часто одно устройство поддерживает сразу несколько форм). Будем рассматривать четыре основные модели организации параллельной вычислительной системы: вычислительные системы с общей и разделенной памятью, векторные системы и графические процессоры общего назначения.

1.2.1. Вычислительная система с разделенной памятью

Как следует из названия, данная система состоит из множества узлов, каждый их которых имеет свою память, недоступную другим узлам. Взаимодействие узлов осуществляется посредством посылки сообщений. Примером таких систем могут служить кластерные суперкомпьютеры (например, кластер ИГЭУ), кластеры типа Beowulf, GRID – системы (проекты типа Folding@Home или SETI@Home). Системы данного класса имеют наиболее высокую пиковую производительность, однако достаточно сложны в программировании и подходят не для всех задач. Основной проблемой является крайне медленное взаимодействие узлов системы (в частности, для GRID-систем оно может осуществляться через Интернет). Для программирования систем с разделенной памятью обычно используют одну из библиотек, реализующих стандарт MPI (например, OpenMPI, MPICH, MS MPI).

1.2.2. Вычислительная система с общей памятью

Примером системы с общей памятью могут служить современные многоядерные процессоры. Каждое ядро представляет собой независимое вычислительное устройство, однако все они разделяют общую память. Это во многом устраняет основную проблему систем с разделенной памятью (скорость межпроцессных взаимодействий), поскольку обращение к памяти на несколько порядков быстрее посылки сообщения по сети, однако количество вычислительных узлов в системе с общей памятью не может быть велико, что связано с техническими трудностями организации общего доступа к памяти. Поэтому пиковая производительность систем с разделенной памятью обычно значительно выше. С другой стороны, многие современные системы сочетают в себе свойства обеих групп. Например, кластер типа Beowulf может состоять из большого количества компьютеров, соединенных сетью, при этом каждый из компьютеров имеет многоядерный процессор и сам по себе является параллельной системой с общей памятью. Для программирования систем с общей памятью существует множество различных библиотек, технологий и языков программирования, однако наибольшее распространение получила технология OpenMP. На наш взгляд, также интерес представляет библиотека TBB (Threading building blocks), разработанная компанией Intel [49].

Параллелизм векторных инструкций. Многие существующие процессоры поддерживают особые векторные инструкции, которые позволяют выполнять за одну операцию некоторое действие сразу над несколькими значениями. В частности, на современных процессорах, используемых в персональных компьютерах, это группы инструкций SSE (SSE, SSE2 SSE3 и т.д.). В результате даже одноядерный процессор способен обрабатывать несколько значений одновременно по принципу SIMD. Данный вид параллельной обработки также прекрасно сочетается с остальными. Например, у упомянутого выше кластера Beowulf, состоящего из множества соединенных сетью компьютеров с многоядерными процессорами, каждое процессорное ядро может быть способно к выполнению векторных инструкций. Единого способа программирования векторных инструкций не существует, поскольку эти инструкции специфичны для конкретного типа процессора. Во многих случаях приходится прибегать к программированию непосредственно на языке ассем-

блера данного процессора либо использовать набор так называемых intrinsic-функций некоторых компиляторов. И то и другое достаточно трудоемко. Однако в некоторых относительно простых ситуациях современные компиляторы (например, Intel C++) способны использовать векторный параллелизм автоматически.

Графические процессоры общего назначения. Современные графические процессоры представляют собой высокопроизводительные вычислительные устройства, зачастую кардинально отличающиеся по архитектуре от центральных процессоров. В частности, задачи обработки графики позволяют организовать массивно-параллельную обработку, т. е. обработку данных большим количеством относительно медленных процессоров. Однако такая схема может быть полезна не только для графики, поэтому существует множество примеров использования графических процессоров для самых разных задач. В последнее время, с выпуском технологий NVIDIA CUDA, AMD FireStream, данное направление приобрело значительную популярность. Можно также отметить технологии OpenCL и DirectCompute, позволяющие абстрагироваться от типа конкретного вычислительного устройства.

1.2.3. Параллельные методы решения ОДУ

Классификация параллельных алгоритмов для решения ОДУ. Согласно Гиру [40], можно выделить три основных способа распараллеливания алгоритмов интегрирования ОДУ:

1. Параллелизм на уровне метода, т. е. выполнение некоторых шагов метода параллельно. Данный вид параллелизма обычно не дает большого ускорения и плохо масштабируется, так как большинство методов состоят из относительно небольшого числа шагов, причем не все они могут быть выполнены параллельно.
2. Параллелизм на уровне системы уравнений предполагает разделение системы на части некоторым образом, так что каждая часть может быть вычислена относительно независимо.
3. Параллелизм по времени, т. е. параллельное вычисление решения на различных временных интервалах.

Как уже было сказано, параллелизм на уровне метода не может обычно дать большого ускорения, а его масштабируемость ограничена небольшим количеством процессоров, однако полностью игнорировать такой подход, на наш взгляд, не стоит. На первый взгляд, хорошими кандидатами на параллельную реализацию являются методы Рунге–Кутты. Хотя классический метод Рунге–Кутты 4-го порядка (РК-4) не может быть распараллелен, поскольку каждое промежуточное значение требует вычисления предыдущего, существуют методы Рунге–Кутты, для которых по крайней мере некоторые промежуточные значения могут быть вычислены независимо.

Возникает вопрос, можно ли построить такой метод Рунге–Кутты, где количество независимо вычисляемых промежуточных результатов достаточно велико. Ответ на него дает следующая теорема [60].

Теорема (Исерлес, Нёрсетт). Для метода Рунге–Кутты количество последовательно вычисляемых стадий (то есть стадий, которые не могут быть вычислены параллельно) не меньше порядка точности метода.

Соответственно, можно сказать, что такой способ распараллеливания может дать сколько-нибудь существенные результаты только для методов Рунге–Кутты высокого порядка точности, поскольку для методов порядка точности выше пятого количество стадий ограничивается снизу так называемыми барьерами Бутчера. Но и в этом случае ожидаемое ускорение вычислений относительно невелико.

Рассмотрим другой подход к этой проблеме. В работе [60] рассматриваются так называемые параллельно-итерируемые методы Рунге–Кутты (PIRK, parallel-iterated Runge–Kutta methods). Суть этих методов заключается в том, что вместо явного решения систем алгебраических уравнений на каждом шаге применяется метод последовательных приближений:

$$K_i^{[0]} = y_n, i = 1, ..., s,$$

$$K_{n+1}^{[\xi]} = y_n + h \sum_{j=1}^{s} a_{ij} f(t_n + c_j h, K_j^{[\xi-1]}), \xi = 1, 2, ...,$$

$$y_{n+1}^{[\xi]} = y_n + h \sum_{j=1}^{s} b_j f(t_n + c_j h, K_j^{[\xi]}).$$

Здесь верхний индекс в квадратных скобках показывает номер итерации. Можно показать (см. [60]), что $y_{n+1}^{[\xi]}$ является аппроксимацией $y(t+h)$ порядка $min(\xi 1, p^*)$, где p^* – порядок исходного неявного метода Рунге–Кутты. Как видно из приведенных выражений, итерации могут выполняться параллельно для каждого i, т. е. масштабируемость такого метода ограничена s процессорами. К тому же итерируемый метод обладает значительно меньшей устойчивостью, чем исходный неявный метод, в частности свойства A- и L-устойчивости теряются.

Методы waveform relaxation. Основная идея этих методов заключается в том, что решение системы из n уравнений заменяется на итерационное решение n систем из одного уравнения [64]. Итерации могут быть организованы различными способами, соответствующими итерационным схемам, применяемым при решении СЛАУ. Если мы используем итерации аналогично методу Якоби, то такой метод носит название Jacoby waveform relaxation. Итерационная процедура этого метода имеет следующий вид:

$$\begin{pmatrix} y_0^{[i+1]}(t) \\ ... \\ y_{n-1}^{[i+1]}(t)... \end{pmatrix} = \begin{pmatrix} f_0(t, y_0^{[i+1]}(t), y_1^{[i]}(t), ..., y_{n-1}^{[i]}(t)) \\ ... \\ f_{n-1}(t, y_0^{[i]}(t), y_1^{[i]}(t), ..., y_{n-1}^{[i+1]}(t)) \end{pmatrix}.$$

Из формулы видно, что каждое из n уравнений может быть решено независимо, соответственно, при параллельной реализации требуется всего один обмен данными между процессами на каждом шаге. Недостатком такого подхода является достаточно медленная сходимость итераций. Скорость сходимости можно увеличить, используя итерации по типу метода Гаусса–Зейделя. К сожалению, полученный метод также имеет серьезный недостаток – он гораздо труднее распараллеливается, чем метод Якоби.

Параллелизм на уровне векторных операций. Выше мы рассматривали способы распараллеливания, специфичные для методов решения систем ОДУ. Однако иногда эти методы неприменимы либо дают недостаточный эффект, в таких случаях стоит обратить внимание на «общие» методы распараллеливания, такие как параллельная реализация векторных операций. Идея заключается в том, что многие численные алгоритмы можно предста-

вить в виде операций над векторами и матрицами и разделить эти векторы и матрицы между процессами. То есть для машины с разделенной памятью в памяти каждого из n процессов находятся соответствующие N/n компонент каждого из используемых в алгоритме N-мерных векторов. Каждый процесс выполняет одну и ту же программу для своего набора данных. Обмен данными происходит при следующих операциях: вычислении правой части системы, вычислении максимального и минимального элементов вектора, вычислении нормы вектора, вычислении скалярного произведения векторов и некоторых других, реже встречающихся. Для большинства операций, таких как сложение векторов, умножение вектора на скаляр, копирование векторов и так далее, межпроцессные коммуникации не требуются. В соответствии с принципами объектно-ориентированного программирования, операции над векторами можно инкапсулировать в отдельном классе, представляющем собой абстракцию вектора. Либо, если язык программирования не поддерживает ООП, предоставить набор BLAS-подобных функций. Рассмотрим реализацию некоторых методов данного класса. Первый пример – реализация метода fill, заполняющего все компоненты вектора константным значением. Легко видеть, что такая операция не требует межпроцессных коммуникаций и каждый процесс может заполнять свою часть массива data вне зависимости от других:

```
void fill(realtype r){
        assert(!data.empty());
        for (unsigned int i = 0; i < local_length; ++i){
                data[i]=r;
        }
}
```

Второй пример – метод, вычисляющий скалярное произведение векторов. Этот метод в отличие от предыдущего требует осуществления обмена информацией между процессами:

```
realtype dot(const mvector& v2) const{
        assert(!data.empty());
        realtype sum=0;
        for (unsigned int i = 0; i < local_length; ++i){
                sum+=data[i]*v2[i];
        }
        realtype globsum;
```

```
        mpi.allreduce(&sum,&globsum,1,mparallel::sum);
        return globsum;
}
```

Здесь mpi – вспомогательный класс для работы с функциями MPI. Метод allreduce всего лишь предоставляет более удобный интерфейс для использования функции MPI_Allreduce:

```
template<class T>
int allreduce(const T* in,
T* out, int cnt, operation op ) const
{
        return MPI_Allreduce((void*)in,(void*)out,
                        cnt,mpi_datatype<T>::value,
                        op,mpi_comm);
}
```

Для машин с общей памятью более выгодной может являться другая реализация. Можно разместить векторы целиком в общей памяти и сделать каждый процесс ответственным за обработку какого-либо конкретного участка. Преимуществом данного способа является отсутствие необходимости в межпроцессных коммуникациях. Для примера рассмотрим возможную реализацию функции dot с использованием OpenMP:

```
realtype dot (const mvector& v2) const {
        realtype sum = 0;
        #pragma omp parallel for reduction(+:sum)
        for (int i=0;i<data.size();++i){
                sum += data[i]*v2[i];
        }
        return sum;
}
```

Аналогичный подход может быть применен для SIMD-систем. Например, при помощи SIMD-расширений современных x86-совместимых процессоров можно реализовать функцию dot таким образом, что за одну операцию вычисляется результат сразу для четырех (в случае одинарной точности) или двух(в случае двойной) компонентов векторов. Исходный код такой реализации на языке ассемблера x86 с использованием SIMD-расширений SSE3 можно найти в [12].

Параллельное решение систем алгебраических уравнений. При решении системы ОДУ неявными методами большая часть времени уходит на решение системы алгебраических уравнений, как правило, нелинейной (хотя, например, в случае методов Розенброка – линейной). Например, в работе [16] приводится сравнение относительного времени выполнения этапов метода Гира в реализации CVODE. Около 80 % времени решения занимает решение системы нелинейных алгебраических уравнений, которое выполняется сочетанием метода Ньютона с итерационным методом решения СЛАУ (BiCG-Stab). Для более простых методов (например, метода трапеций) это соотношение может еще возрасти. Поэтому даже последовательную реализацию неявного метода решения системы ОДУ часто можно ускорить, распараллелив метод решения системы алгебраических уравнений. Существует два основных подхода к параллельной реализации этих методов. Первый подход – применение блочных методов. Блочная реализация возможна как для линейных, так и для нелинейных систем. Линейные блочные методы (блочные методы Якоби и Гаусса–Зейделя) подробно описаны в работе [53]. Для нелинейных систем блочная реализация метода Ньютона может выглядеть, например, следующим образом [30]. Разобьем систему на перекрывающиеся блоки, то есть соседние блоки будут иметь несколько общих элементов. Решение системы линейных алгебраических уравнений $J_n \triangle x_{n+1} = -f(x_n)$ находится независимо каждым блоком (на этом этапе выполнение каждого блока не зависимо от других, поэтому каждый блок может выполняться отдельным процессом), затем блоки обмениваются информацией о значениях граничных элементов, принимая за новое значение некоторую линейную комбинацию соответствующих значений соседних блоков. Данная схема обладает достаточно высокой степенью параллелизма, однако часто существенно увеличивает количество итераций, необходимых для достижения сходимости. Существует множество способов ускорения сходимости методов такого рода, описание которых выходит далеко за рамки нашей работы. Второй подход к распараллеливанию методов решения систем нелинейных алгебраических уравнений (точнее, метода Ньютона) основан на наблюдении, что большая часть вычислительной нагрузки при использовании этого метода ложится на этап решения СЛАУ. Из этого следует, что путем использования параллельного алгоритма решения СЛАУ мы можем значительно повысить производи-

тельность метода Ньютона в целом. Как уже было отмечено, наиболее успешным в этом смысле является использование методов Ньютона–Крылова, т. е. методов Ньютона, в которых решение линейной системы производится одним из методов Крылова. Особенно эффективны методы Нютона–Крылова для решения больших разреженных систем с нерегулярной структурой разреженности, т. е. одного из наиболее сложных для решения классов систем, тем не менее часто встречающегося на практике. Параллельная реализация методов Крылова достаточно проста: весь алгоритм можно представить в виде последовательности операций линейной алгебры, причем, помимо умножения матрицы на вектор, все операции осуществляются над парами векторов. Параллельная реализация операций между векторами не представляет сложности, умножение матрицы на вектор также в большинстве случаев может быть достаточно просто распараллелено, однако при этом могут возникать сложности, связанные с размером и разреженностью матрицы. Интерес представляет реализация на GPU метода итераций Чебышёва, так как этот метод не требует операции скалярного произведения и, соответственно, глобальной синхронизации потоков, представляющей для GPU определенную проблему. Однако недостатком этого метода является необходимость априорного знания характеристик спектра системы, в частности, оценки величины главного собственного значения. Несколько вариантов реализации этого метода рассмотрены в работе [13].

1.3. Выводы

Для решения жестких систем ОДУ разработано множество методов, однако эта задача, тем не менее, представляет определенную сложность ввиду следующих соображений:

- Большинство методов интегрирования жестких систем являются неявными, следовательно требуют существенно больших вычислительных затрат, чем используемые для решения нежестких систем явные методы. В случае большой размерности системы эти затраты могут сделать решение невозможным или потребовать использования суперкомпьютеров.
- Существует группа явных методов, предназначенных для интегрирования жестких систем. Эти методы имеют существенное преимущество в

производительности, однако обладают невысокой точностью и ограниченной устойчивостью. Поэтому, вопрос применимости явных методов к тому или иному классу систем требует отдельного исследования.

– Проблема производительности численного интегрирования решается методами параллельного программирования. В частности, одним из современных направлений является использование графических процессоров общего назначения. Однако, ввиду особенностей численных методов и архитектуры многопроцессорных вычислительных систем, параллельная реализация численного интегрирования представляет собой нетривиальную задачу и также требует отдельного исследования.

Таким образом, для создания программного комплекса для численного интегрирования выбранного класса жестких систем, обеспечивающего высокую производительность, необходимо провести исследование применимости для подобных систем явных методов с расширенной областью устойчивости и создать программную реализацию подходящего метода с использованием графических процессоров общего назначения. В настоящий момент не существует широко распространенных программных реализаций подобного подхода, поэтому создание такого комплекса является актуальной задачей.

2. Реализация проекционного метода на графическом процессоре общего назначения

2.1. Архитектура массивно-параллельной вычислительной системы на примере GPGPU NVIDIA

2.1.1. Преимущества архитектуры GPU

В последнее время возрастает интерес к возможности переноса части вычислений на графические процессоры общего назначения (GPGPU), такие как NVIDIA GeForce и Tesla и AMD Radeon. Производители этих устройств предоставляют API для их использования в качестве процессоров общего назначения (в частности, в данной работе используется API NVIDIA CUDA) и заявляют крайне высокие значения пиковой производительности, до 1 TFlops на GetForce GTX 285. Для иллюстрации, будет уместно привести график из [52], показывающий соотношение пиковой производительности современных центральных и графических процессоров (рисунок 2.1).

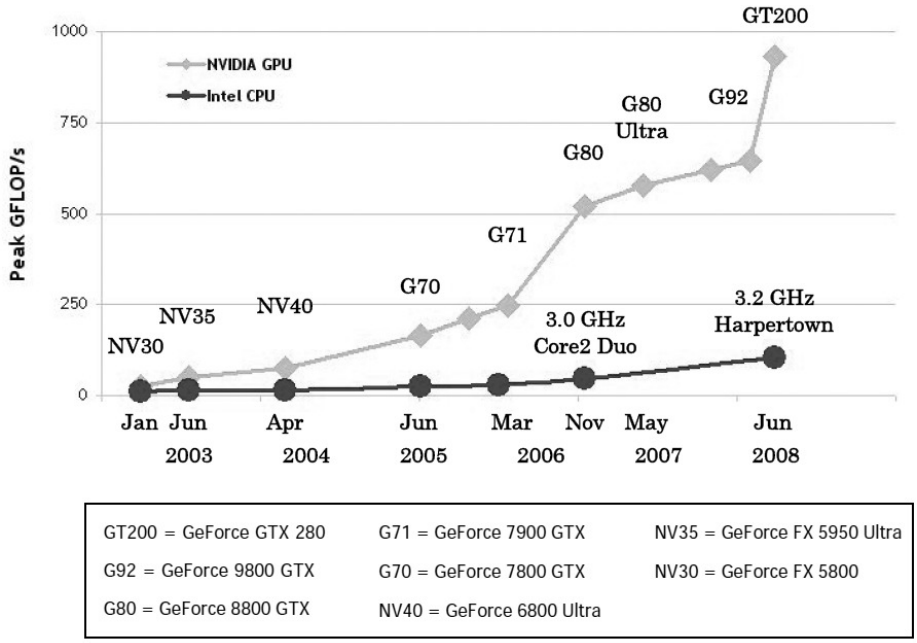


Рисунок 2.1. Производительность CPU и GPU

2.1.2. Архитектура на примере GPU NVIDIA

Архитектура GPGPU во многом отличается от архитектуры классических параллельных вычислительных систем. Мы будем рассматривать системы NVIDIA Tesla, однако большая часть сказанного применима и к GPGPU других производителей. Каждое устройство NVIDIA Tesla (device в терминологии NVIDIA [52]) представляет собой сопроцессор, работающий в связке с центральным процессором (host). С аппаратной точки зрения устройство имеет большое количество (240 для Tesla C1060) так называемых потоковых процессоров. С программной точки зрения, устройство способно выполнять множество потоков, объединенных в один или несколько блоков (рисунок 2.2). В отличие от классических параллельных вычислительных систем, в которых создание и поддержание каждого потока требует достаточно значительных накладных расходов, аппаратный планировщик графического процессора может эффективно работать с сотнями и тысячами потоков. Поэтому стандартным подходом к решению задач на GPU является выделение отдельного потока на каждый элемент данных. Блоки, в свою очередь, объединяются в сетку (grid), которая может быть одно-, двух- и трехмерной. Аппаратный планировщик потоков распределяет блоки между вычислительными устройствами, называемыми мультипроцессорами, объединяющими несколько потоковых процессоров. Потоки внутри каждого блока разделаются на варпы (warp) по 32 штуки и выполняются в каждый момент времени на одном мультипроцессоре по принципу SIMD. Функция, исполняемая на устройстве, называется ядром (kernel). Синхронизация потоков возможна только внутри одного блока. Если требуется синхронизация нескольких блоков между собой, то это может быть сделано (часто с существенной потерей производительности) путем разбиения ядра на несколько частей и последовательного их вызова. Иерархия памяти GPU NVIDIA следующая [52]:

1. Глобальная (global) память. Имеет большой объем и доступна из всех блоков, однако скорость доступа к ней крайне низкая: время чтения элемента из глобальной памяти в несколько сотен раз превышает типичное время арифметической операции.
2. Локальная (local) память. Доступна только выделившему потоку. Скорость работы сравнима с глобальной, поэтому на практике обычно используется только в случае нехватки более быстрой памяти.

3. Разделяемая (shared) память. Доступна всем потокам блока. Скорость доступа высокая, порядка времени выполнения арифметических операций, однако для каждого блока доступно всего не более 16 Кб разделяемой памяти.
4. Регистры (register). Доступны только текущему потоку, скорость доступа высокая, однако количество ограничено.
5. Константная (constant) память. Доступна всем блокам для чтения. Данный вид памяти кэшируется, поэтому первое обращение может выполняться относительно медленно (сравнимо с глобальной памятью), а последующие, в случае, если значение берется из кэша, значительно быстрее.
6. Текстурная (texture) память. Аналогична константной по модели доступа, отличаясь от нее по стратегии кэширования. Подробнее, см. [52].

Немаловажной особенностью устройства является то, что основным типом данных является число с плавающей точкой одинарной точности, причем реализованное с некоторыми отклонениями от стандарта IEEE-754, числа с двойной точностью также поддерживаются на более новых моделях GPU, однако со значительно меньшей производительностью.

Стоит отметить, что для GPGPU «узким местом», как правило, является доступ к глобальной памяти (в [52] утверждается, что обращение к глобальной памяти имеет латентность в 400-600 тактов), однако при выполнении определенных условий 16 запросов к памяти от различных потоков могут быть объединены (coalesced) в одну операцию обращения к памяти. Условия, при которых может произойти такое объединение, достаточно сложны и отличаются для различных поколений графических процессоров, однако нужно иметь в виду, что минимизация количества необъединенных обращений к глобальной памяти приводит в большинстве случаев к значительному приросту производительности.

Применение GPGPU для решения систем ОДУ часто требует определенной изобретательности. Действительно, наиболее ресурсоемкая операция вычисления правой части системы уравнений в общем случае плохо ложится на архитектуру SIMD, поскольку каждое уравнение может вычисляться совершенно особым образом. К тому же применение GPGPU эффективнее применения CPU только в случае, если алгоритм удается распараллелить на

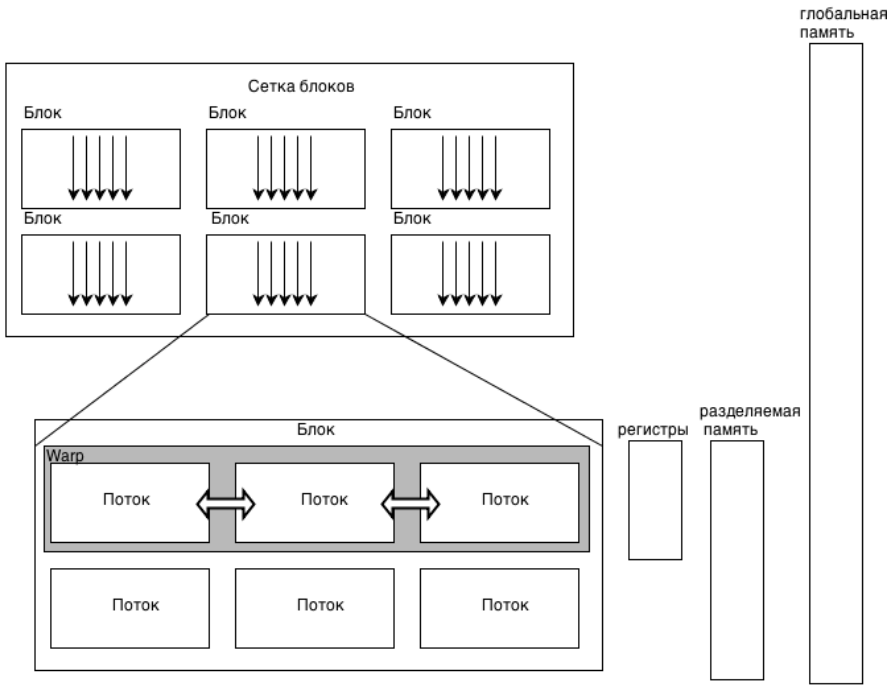


Рисунок 2.2. Модель исполнения CUDA

несколько сотен, а лучше тысяч потоков, что при обычном подходе не всегда осуществимо. С другой стороны, существует множество систем, правая часть которых представляет собой небольшое количество групп, одинаковых с точностью до коэффициентов уравнений. В таком случае все уравнения группы могут выполняться в SIMD-блоке, и реализация на графическом процессоре может оказаться крайне эффективной. Если такое разбиение системы на блоки произвести не удается, остается другая возможность ускорения вычислений при помощи графических процессоров – перенос на них решения систем алгебраических уравнений. Такой подход может оказаться достаточно эффективным, например в работе [14] приводится реализация метода BiCG-Stab, подходящего для реализации метода типа Ньютона–Крылова. Приведенная реализация показывает в 7 раз большую производительность, чем четырехъядерный центральный процессор, при том что возможности ее ускорения как экстенсивным (наращивание количества процессоров), так и интенсивным путем (оптимизация алгоритма, использование более качественного предобусловливания) не исчерпаны. Если заведомо известно, что матрица линейной системы будет симметричной положительно-определенной, то вместо BiCG-Stab можно применить более простой метод сопряженных градиентов, реализация которого для GPU описана в работе [26]. Для плотных систем может оказаться эффективным использование одного из параллель-

ных вариантов метода LU-разложения. Реализация LU-разложения на GPU приведена в работе [62]; данная реализация имеет пропускную способность 179 GFlop/s, что втрое превышает производительность современного четырехъядерного CPU.

2.2. Общая структура параллельной реализации

Рассмотрим реализацию одного из явных методов с расширенной областью устойчивости, а именно проекционного метода Эйлера, на графическом процессоре общего назначения. Из описания архитектуры графических процессоров общего назначения NVIDIA [52] следует, что они получают преимущество перед центральным процессором в том случае, если задачу можно разбить на большое количество параллельных подзадач. Таким образом, задачи невысокой размерности (до сотен уравнений) решать на GPU, как правило, неэффективно. Для задач высокой размерности можно выделить две сферы применения графических процессоров: распараллеливание вычисления правой части системы, а также, для неявных методов, распараллеливание решения получаемой системы алгебраических уравнений. Другие источники параллелизма при решении систем ОДУ (например, параллельное исполнение шагов метода Рунге-Кутты), как правило, не играют большой роли при использовании GPU, так как не способны обеспечить распараллеливание на сотни и тысячи потоков. Соответственно, для явного метода основное ускорение при параллельной реализации программы интегрирования большой системы ОДУ на массивно-параллельной вычислительной системе получается от параллельной реализации вычисления правой части системы, соответственно такая программная реализация будет отличаться для каждой решаемой системы. Тем не менее возможно выделить несколько общих приемов, пригодных для использования при решении широкого класса задач. Первое, что следует указать - стратегию распараллеливания, т. е. распределение данных по потокам. Графический процессор является системой с общей памятью, однако иерархия памяти устроена таким образом, что общая память является достаточно медленным ресурсом, а более быстрая память доступна только на уровне блока или одного потока. Следовательно, для системы дифференциальных уравнений разумным является «горизонтальное» разбиение, когда каждый поток вычисляет одно или более уравнений, но каждое урав-

нение вычисляется не более чем одним потоком. Тогда систему ОДУ можно представить в виде

$$\frac{dx}{dt} = F(f_l(t, x_k...) + f_s(x, t, ...)),$$

где f_l - некоторая функция от локальных относительно уравнения переменных, т. е. для k - го уравнения в системе f_l не зависит от $x_i \forall i \neq k$; f_s - функция от глобальных переменных, то есть требующая для вычисления данных других уравнений. В случае, если уравнения распределены по различным процессорам параллельной вычислительной системы, f_l может быть целиком вычислена на текущем процессоре, тогда как f_s требует выполнения одной или нескольких операций редукции. Таким образом, общий алгоритм вычисления правой части системы на параллельной вычислительной системе следующий:

1. Вычислить f_l.
2. Выполнить редукцию, вычислить f_s.

Отметим, что такой порядок вычислений, как правило, требует модификации системы уравнений, поскольку необходимо переупорядочить вычисления таким образом, чтобы операция редукции выполнялась отдельно от остальных операций.

2.3. Реализация операции редукции

Относительно реализации операции редукции, наиболее частым подходом является двухэтапная древовидная редукция, подробно описанная в работе [43]. Однако далее мы покажем, что для нашей системы такой подход является субоптимальным. Причина этого состоит в том, что для явного метода решения системы ОДУ и нашего способа разделения по потокам вычисления правой части накладные расходы, связанные с запуском ядра, составляют существенную долю общего времени выполнения программы. В этом случае естественной оптимизацией является минимизация количества используемых ядер. Поэтому вместо двухэтапной редукции мы будем использовать одноэтапную, построенную на атомарных операциях. Нужно отметить, что необходимые для реализации атомарные операции доступны в устройствах с compute capability 1.1 и выше, однако на устройствах с compute capability

меньше 2.0 производительность их невелика, так что для этих устройств возможно стоит рассмотреть использование двухэтапной редукции.

Основная идея одноэтапной редукции заключается в том, что каждый блок после выполнения первого этапа редукции (редукции внутри каждого блока) атомарно увеличивает значение глобального счетчика. Тот блок, который увеличивает значение счетчика до значения, равного количеству блоков, очевидно является последним и может выполнить второй этап редукции. Более подробно алгоритм редукции приведен на рисунках 2.3 и 2.4, исходный код с подробными комментариями приведен в приложении 1.

Важной особенностью графических процессоров является то, что они реализуют слабую модель памяти [52], следовательно:

- порядок, в котором данные записываются в память потоком, не обязательно совпадает с порядком, в котором эти данные записываются с точки зрения другого потока;
- порядок, в котором данные считываются из памяти, не обязательно совпадает с порядком инструкций в программе.

Это означает, что для корректной работы алгоритма редукции необходимо использование барьера памяти, без которого возможна ситуация, когда изменения в глобальной памяти, выполненные в одном warp, будут недоступны для потоков другого, в результате чего общий результат алгоритма будет неверен.

Функция, которая выполняет редукцию внутри блока, имеет реализацию, аналогичную таковой в двухэтапном алгоритме, с некоторыми особенностями. Общая схема алгоритма при этом следующая (см. также рис 2.3):

1. Загрузить данные в разделяемую память.
2. Синхронизация потоков внутри блока.
3. s - количество элементов в массиве.
4. Пока s>0
   - 4.1. Каждый из $\frac{s}{2}$ потоков складывает по два элемента в разделяемой памяти.
   - 4.2. Если $\frac{s}{2} \geq 32$, выполнить синхронизацию внутри блока.
   - 4.3. $s = \frac{s}{2}$.
5. Записать результирующее значение в глобальную память.

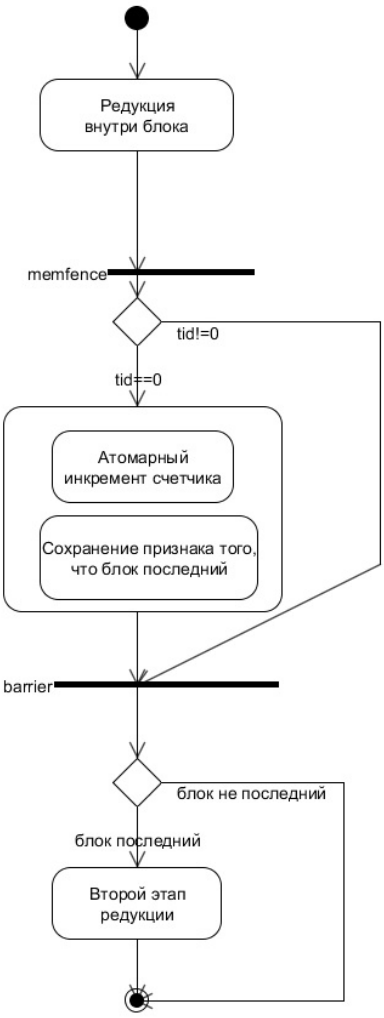


Рисунок 2.3. Диаграмма алгоритма редукции

В неизмененном виде такой алгоритм может работать только в случае длины массива, являющейся степенью двойки, чем и обусловлен наш выбор количества уравнений для тестовой задачи. В работе [43] приведено 7 оптимизаций, позволяющих поднять производительность по сравнению с «наивной» реализацией примерно в 25 раз. Большая часть этих оптимизаций применима и в нашем случае, за исключением тех, что уменьшают количество требуемых потоков, так как у нас количество потоков фиксировано и задается реализацией функции вычисления правой части. Исходный код функции с подробными комментариями приведен в приложении 1, однако следует отметить некоторые важные технические особенности, важные для обеспечения высокой производительности:

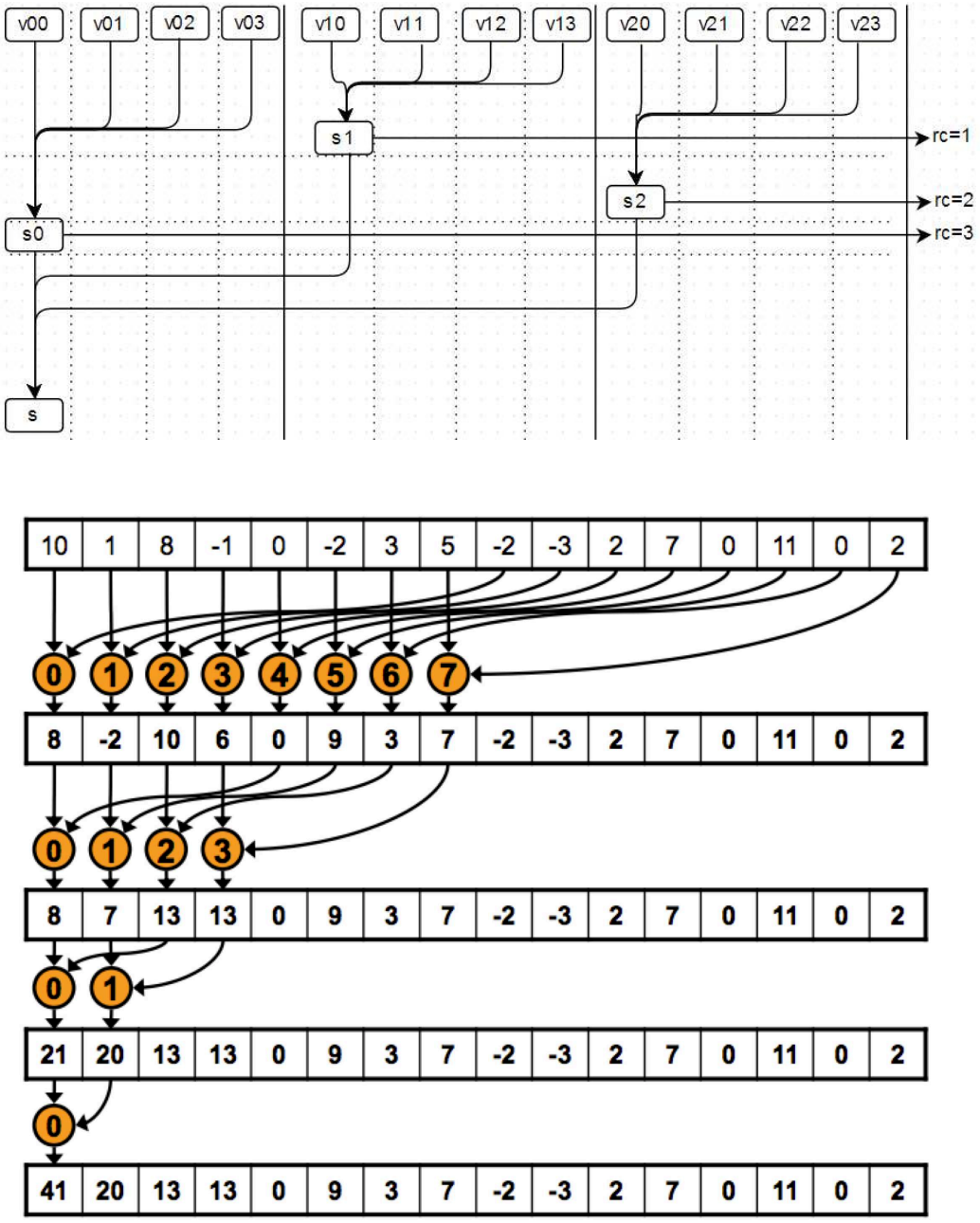


Рисунок 2.4. Операция редукции

– алгоритм реализован в форме шаблонной функции поддерживаемого CUDA диалекта языка C++. Это позволяет вынести ряд ветвлений, существенно снижающих производительность любой массивно-параллельной вычислительной системы, на этап компиляции;
– циклы максимально развернуты, что также обеспечивает минимальное количество ветвлений;
– после каждого раунда сложения элементов требуется синхронизация внутри блока, однако если оставшиеся рабочие потоки входят в один warp, они выполняются синхронно и дополнительная синхронизация не требуется;

2.4. Реализация метода интегрирования

Воспользовавшись изложенными выше соображениями, можно написать программу интегрирования системы ОДУ на графическом процессоре общего назначения. В начале рассмотрим реализацию простейшего метода - явного метода Эйлера. Реализация этого метода на GPU достаточно тривиальна: необходимо вызвать процедуру вычисления правой части системы, а затем каждый элемент результата умножить на длину шага. Для каждого шага по времени будем вызывать отдельное ядро, таким образом мы обеспечим синхронизацию. Исходный код одного шага метода Эйлера будет иметь следующий вид:

```
__global__ void ode_step1_euler(RealT t,
RealT h, RealT* x, RealT *tmp,
int sz,
void* additional_data){
  F(t,x,tmp,sz,additional_data);   //tmp = F(t,x)
  sz/=3;
  axpy3(h,x,tmp,sz);           //x = h*tmp + x
  __threadfence();
  __syncthreads();

  //редукция
  DeviceTurbData* data_gpu = (DeviceTurbData*)additional_data;
  float* reduced_ptr = x;
  float* reduce_buffer = data_gpu->reduce_buffer;
  const int reduce_size = sz;
  reduce_device(reduced_ptr,reduce_buffer,reduce_size);
}
```

Проекционный метод Эйлера включает в себя несколько (мы используем 4) внутренних шагов и один шаг интерполяции. В нашей реализации внутренние шаги осуществляются явным методом Эйлера. Простейшая реализация проекционного метода может работать по следующей схеме:

1. В цикле вызвать 4 раза ядро интегрирования методом Эйлера (включает в себя вычисление правой части и, соответственно, редукцию).

2. Запомнить два последних решения в области глобальной памяти.

3. Вызвать ядро вычисления шага интерполяции.

Такой подход имеет право на существование, более того, превосходит по производительности реализацию на центральном процессоре. Однако такая реализация включает в себя высокие накладные расходы на большое количество вызовов функций-ядер, что во многих случаях может привести к тому, что накладные расходы составят большую часть времени вычислений, вследствие чего общая эффективность решения будет крайне низкой. Используя алгоритм редукции с одним вызовом ядра, накладные расходы можно существенно (теоретически, в 4 раза) уменьшить. Это возможно в случае, если операции вычисления правой части, интерполяции и редукции используют одинаковое количество потоков (см. рисунок 2.5).

Схема программы, реализующей проекционный метод Эйлера на GPU, приведена на рисунке 2.6.

Для современных компьютеров оптимальная работа с памятью является одним из важнейших требований для обеспечения высокой производительности. Причиной является тот факт, что обращение к оперативной памяти требует на порядок больше времени, чем арифметические операции процессора. По этой причине подсистема памяти включает в себя многоуровневую иерархию кэшей, а также сложные алгоритмы предвыборки и пакетной передачи данных. Все эти механизмы работают прозрачно для пользователя, тем не менее для высокопроизводительных алгоритмов желательно учитывать их влияние. Таким образом, важное значение имеет расположение данных в памяти. Очевидно, что для реализации алгоритмов интегрирования на различных устройствах требуются различные формы расположения данных в памяти. Обозначим $x^{k,0}, x^{k,1}, ...$ переменные, вычисляемые k-м потоком. Для однопоточной реализации наиболее эффективен простейший вариант расположения данных последовательно: $[x_0^{0,0} \dots x_{N-1}^{0,0}, x_0^{0,1} \dots x_{N-1}^{0,1}, ...]$. Из теорети-

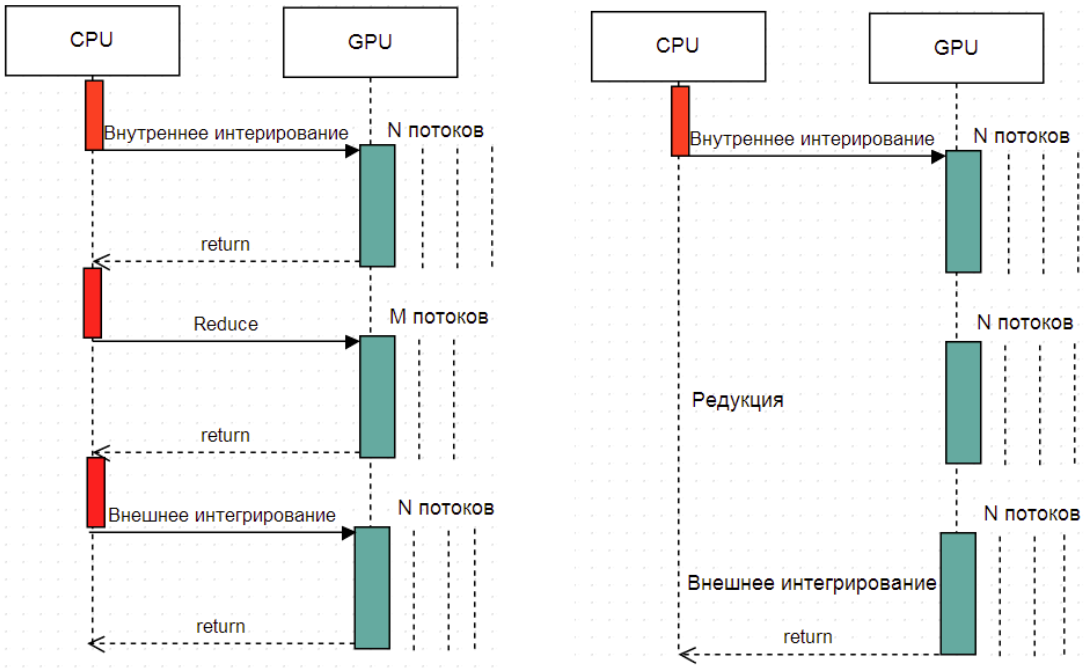


Рисунок 2.5. Реализация проекционного метода Эйлера на GPU

ческих соображений можно предположить, что возможно найти лучший вариант, однако потенциальный выигрыш в производительности будет невелик, а оптимизация однопоточной версии выходит за рамки данной работы. Для многопоточной версии такое представление очевидно не будет оптимальным. Действительно, даже если отбросить соображения по расположению данных по кэш-линиям, многопоточная реализация программы интегрирования на центральном процессоре в любом случае выделяет на каждый поток некоторый блок из множества уравнений (так как число физических процессоров, имеющих общую память, невелико, значительно меньше числа уравнений, случай машин с разделенной памятью мы не рассматриваем). Если данные расположены в памяти как указано выше, то мы лишаемся важной возможности - каждому потоку простым образом выполнить вычисления над своим блоком памяти. Таким образом, более разумным выглядит следующее рас-

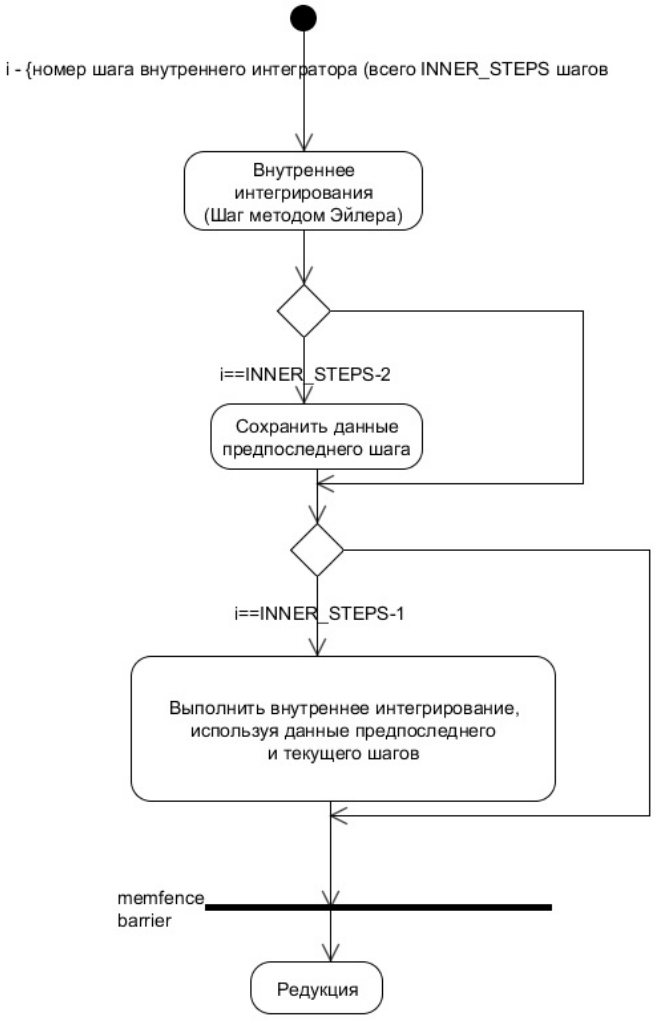


Рисунок 2.6. Диаграмма программной реализации проекционного метода Эйлера на GPU

положение данных:

$$[[x_0^{k,0} \dots x_{b_0-1}^{k,0}, x_0^{k,1} \dots x_{b_0-1}^{k,1}, \dots],$$
$$[x_{b_0}^{k,0} \dots x_{b_1-1}^{k,0}, x_{b_0}^{k,1} \dots x_{b_1-1}^{k,1}, \dots],$$
$$\dots,$$
$$[x_{b_{n-2}}^{k,0} \dots x_{b_{n-1}-1}^{k,0}, x_{b_{n-2}}^{k,1} \dots x_{b_{n-1}-1}^{k,1}, \dots]] \quad ,$$

где $n-$ количество процессоров, $i-$й процессор обрабатывает блок длины $b_i - b_{i-1}$, $b_{-1} = 0$. Такое расположение имеет следующие преимущества:

– простая и эффективная реализация каждым потоком операций над своим блоком данных;

– увеличение локальности данных: данные, используемые каждым потоком, физически расположены по соседним адресам, что упрощает работу менеджера кэша.

Для GPU ситуация будет иной: они аппаратно оптимизированы под большое количество потоков. Менеджер памяти GPU оптимизирован для случая, когда потоки с последовательными номерами обращаются к последовательным ячейкам памяти, поэтому приведенная выше оптимизация для графического процессора, скорее всего, уменьшит производительность. Таким образом, мы сохраним следующее расположение данных в памяти:

$$[x_0^{k,0} \dots x_{N-1}^{k,0}, x_0^{k,1} \dots x_{N-1}^{k,1}, ...].$$

Для наборов констант α, η и др., разумно использовать текстурную память GPU, преимуществом которой является специальная стратегия кэширования, однако вычислительные эксперименты показывают, что иногда накладные расходы на инициализацию текстурной памяти превышают получаемый выигрыш по скорости доступа, поэтому в рассматриваемых далее вычислительных экспериментах используется вариант программы, где константы находятся в глобальной памяти.

2.5. Выводы

В данной главе было показано, что для программной реализации проекционного метода Эйлера на графическом процессоре общего назначения существует ряд техник, не зависящих от конкретной системы уравнений, но при этом позволяющих добиться оптимизации вычислений:

– уменьшение накладных расходов на обращение к графическому процессору путем уменьшения количества вызовов ядер;
– специальная реализация операции редукции;
– использование низкоуровневых оптимизаций (уменьшение ветвлений, использование разделяемой памяти и т.д.).

При этом полное использование потенциала графического процессора при решении системы ОДУ невозможно без параллельной реализации вычисления правой части системы, для исследования которого требуется выбрать некоторую модельную систему.

3. Модель системы

3.1. Модельная система уравнений

Многие технические системы (электрические, гидравлические, акустические и другие) описываются системами уравнений, подобными рассмотренной ниже. Пусть в некоторой сети циркулирует некоторая субстанция, характеризующаяся величиной Y, причем для Y выполняются законы Кирхгофа. Рассмотрим некоторый источник, характеризующийся параметром x и генерирующий Y. Реальными примерами таких систем могут быть электрические цепи, где Y будет представлять собой силу тока, или, например, гидравлические, где аналогом будет расход воды. Параметр x изменяется под действием положительного момента Q и моментов сопротивления, зависящих от самого параметра x, а также от порождаемого значения Y:

$$\frac{dx}{dt} = \gamma Q - \eta x|x| - \xi x Y.$$

Параметр Y вычисляется из уравнения

$$(r + R)Y = \psi x.$$

Здесь, r , R - некоторые константы, характеризующие сопротивление источника и сети.

Работа источника контролируется ПДД-регулятором, уравнение которого имеет вид

$$\frac{d^2\rho}{dt^2} = x^2\rho - \alpha\frac{d\rho}{dt} - k^2(\rho - \rho_0).$$

Здесь ρ - уровень регулирования; k^2 , α - некоторые коэффициенты;ρ_0 - некоторое базовое значение ρ.

Примем, что Q зависит от ρ следующим образом:

$$Q = Q_{max}\frac{e^{-\theta}}{1 + e^{-\theta}},$$

где

$$\theta = \frac{\rho - \rho_{min}}{\rho_{max} - \rho_{min}}\upsilon_1 - \upsilon_2.$$

Коэффициенты υ_1, υ_2 выберем таким образом, чтобы обеспечить достаточно плавное изменение Q в зависимости от ρ на всей области определения, например $\upsilon_1 = 4, \upsilon_2 = 2$.

3.2. Расширение на произвольное количество источников

Для произвольного числа источников, подсоединенных к общей нагрузке, уравнения для x и ρ не изменятся, за исключением того, что в системе теперь будет по паре таких уравнений для каждого источника. Уравнение связи, используемое для вычисления для Y, из законов Кирхгофа:

$$R\sum Y_i = W,$$

где W - значение, характеризующее производительность источника (ЭДС генератора, напор воды, подаваемой насосом и т. д.) и упрощенного предположения, что это значение прямо пропорционально x:

$$\psi_i x_i = w_i.$$

Выразим Y_i и проведем ряд алгебраических преобразований. В результате получим следующее выражение:

$$Y_i = \frac{\frac{\psi_i}{r_i}x_i(\sum\frac{R}{r_j} - 1) - \frac{R}{r_i}\sum\frac{\psi_j}{r_j}x_j}{1 - \sum\frac{R}{r_j}}. \tag{3.1}$$

Итак, общий вид исследуемой системы следующий:

$$\begin{cases} \frac{dx_i}{dt} = \gamma_i Q_i - \eta_i x_i|x_i| - \xi_i x_i Y_i, \\ \frac{d\phi_i}{dt} = x_i^2\rho_i - \alpha_i\frac{d\rho_i}{dt} - k_i^2(\rho_i - \rho_0), \\ \frac{d\rho_i}{dt} = \phi_i, \end{cases} \tag{3.2}$$

где

$$Q_{\mathrm{i}} = Q_{max}\frac{e^{-\theta_i}}{1 + e^{-\theta_i}},$$

$$\theta_i = \frac{\rho_i - \rho_{min}}{\rho_{max} - \rho_{min}}\upsilon_1 - \upsilon_2,$$

$$Y_i = \frac{\frac{\psi_i}{r_i}x_i(\sum \frac{R}{r_j} - 1) - \frac{R}{r_i}\sum \frac{\psi_j}{r_j}x_j}{1 - \sum \frac{R}{r_j}}.$$

3.3. Включение потребителей

Данную систему можно расширить, введя в нее не только источники, а также потребителей, которых можно представить как источники, для которых положительный момент $\gamma Q = 0$. С математической точки зрения для такого преобразования системы требуется лишь положить $\gamma_i = 0$ для всех i, соответствующих потребителям. С вычислительной точки зрения в таком случае уравнения для $\frac{d\phi}{dt}$и $\frac{d\rho}{dt}$ становятся не нужны, поскольку их вычисление влияет только на параметр Q. Учитывая вышесказанное, система 3.2 для N_g источников и N_C потребителей будет выглядеть следующим образом:

$$\begin{cases} \frac{dx_i}{dt} = \gamma_i Q_i - \eta_i x_i|x_i| - \xi_i x_i Y_i & i = 0 \dots N_g, \\ \frac{dx_i}{dt} = -\eta_i x_i|x_i| - \xi_i x_i Y_i & i = N_g \dots N, \\ \frac{d\phi_i}{dt} = x_i^2\rho_i - \alpha_i\frac{d\rho_i}{dt} - k_i^2(\rho_i - \rho_0) & i = 0 \dots N_g, \\ \frac{d\phi_i}{dt} = 0 & i = N_g \dots N, \\ \frac{d\rho_i}{dt} = \phi_i & i = 0 \dots N_g, \\ \frac{d\rho_i}{dt} = 0 & i = N_g \dots N. \end{cases} \tag{3.3}$$

С практической точки зрения подобные системы встречаются во многих прикладных областях. Так, если параметр Y сопоставить с силой тока, параметр S – с ЭДС, а параметр x – с частотой вращения турбины, мы получим энергетическую систему с генераторами постоянного тока, работающими на общую нагрузку. Для гидравлической системы, состоящей из нескольких насосов и гидравлической сети, параметр Y будет представлять собой расход воды, параметр S – напор, структура системы в целом не изменится. Аналогичный прием может быть применен для акустических или полностью механических систем. Таким образом, решение описанной системы позволит найти подход к решению широкого круга прикладных задач в различных областях деятельности.

4. Интегрирование системы

4.1. Интегрирование модели с одним источником

Рассмотрим сначала случай одного источника. В такой конфигурации система содержит три уравнения.

Значения констант выберем так, чтобы установившиеся режимы работы и характерные времена переходных процессов имели тот же порядок величины, что и в реальных системах. Так, будем считать, что время достижения установившегося режима имеет порядок нескольких минут, время остановки под действием механических сил имеет порядок десятков минут. Были получены такие значения: $\gamma = 0,0156$; $\eta = 0,0000323$; $\xi = 0,0001$; $\alpha = 5$; $k^2 = 520$; $\psi = 12$; $\rho_{min} = 3$; $\rho_{max} = 10$. Режим работы системы следующий: в момент времени $t = 0$ положительный момент отсутствует; далее он появляется и в момент времени t_1 подключается нагрузка.

Для решения будем использовать следующие методы:

1. Метод Рунге–Кутты 4-го порядка [65].
2. Метод Богацки–Шампайна 3-го порядка [25].
3. Метод Дормана–Принса 5-го порядка [33, 63].
4. Метод Эйлера–Коши(также известен как метод Хойна) [2].
5. Неявный метод Эйлера [2].
6. Метод Трапеций [2].
7. Метод Адамса–Башфорта–Моултона [2].

Как можно видеть, в списке представлены методы трех разных групп: явные методы типа Рунге–Кутты 2,3,4,5 порядков точности (методы 1–4), неявные методы (5 и 6) и метод типа предиктор-корректор (6). Для неявных методов решение получаемой системы алгебраических уравнений производилось методом Ньютона, использующим, в свою очередь, для решения системы линеаризованной системы метод LUP–разложения [31]. Будем интегрировать данную систему уравнений с различным шагом и рассмотрим зависимость точности вычислений и затрат машинного времени для каждого метода. Под точностью будем понимать rooted-mean square разности между полученным решением и приближением точного решения, полученным интегрированием системы с очень малым шагом. Заметим, что использование в практических

целях интегрирования с таким размером шага, который мы принимаем за точное решение, малоэффективно, поскольку время вычислений даже в рассматриваемых нами относительно простых условиях (один источник, интервал интегрирования 10 минут), вычисления занимают достаточно много времени. Как можно видеть из графиков (рисунок 4.1), поведение методов при изменении шага значительно отличается. Так, только неявными методами удалось успешно решить задачу при всех использованных значениях шага. При этом если неявный метод Эйлера показывает практически линейную зависимость точности от размера шага, то для метода трапеций эта зависимость носит достаточно сложный характер, его точность в некоторых случаях ухудшается при уменьшении шага. С точки зрения времени выполнения зависимость обратная – неявные методы ожидаемо значительно уступают явным в быстродействии, причем это отставание возрастает с уменьшением шага достаточно быстро. Интересен тот факт, что метод Богацки–Шампайна, имеющий 3-й порядок точности, в большинстве случаев превосходит методы Рунге-Кутты-4 и Дормана-Принса, имеющие соответственно 4-й и 5-й порядок точности при меньших затратах машинного времени. Более подробное обсуждение результатов решения подобной системы приведено в работе [17]. Для нас важно следующее: в подобной конфигурации наиболее эффективными являются классические явные методы, соответственно данная система не является жесткой.

4.2. Интегрирование модели с несколькими источниками

Рассмотрим теперь систему с несколькими источниками. Для примера возьмем количество источников $N = 20$ и попробуем решить ее классическим методом Рунге–Кутты 4-го порядка. Вычисления показывают, что максимальный шаг, при котором обеспечивается приемлемое значение точности равен 0,04с, что при промежутке интегрирования в 1000с требует 100000 вычислений правой части. Учитывая, что в рассматриваемой системе основным интересующим нас с практической точки зрения является параметр x, вычисление которого и является целью моделирования системы, и этот параметр в силу инерционности источника изменяется достаточно медленно, мы можем сделать предположение, что шаг в 0,04 с является слишком малым с практической точки зрения, и система скорее всего является жесткой. Мы

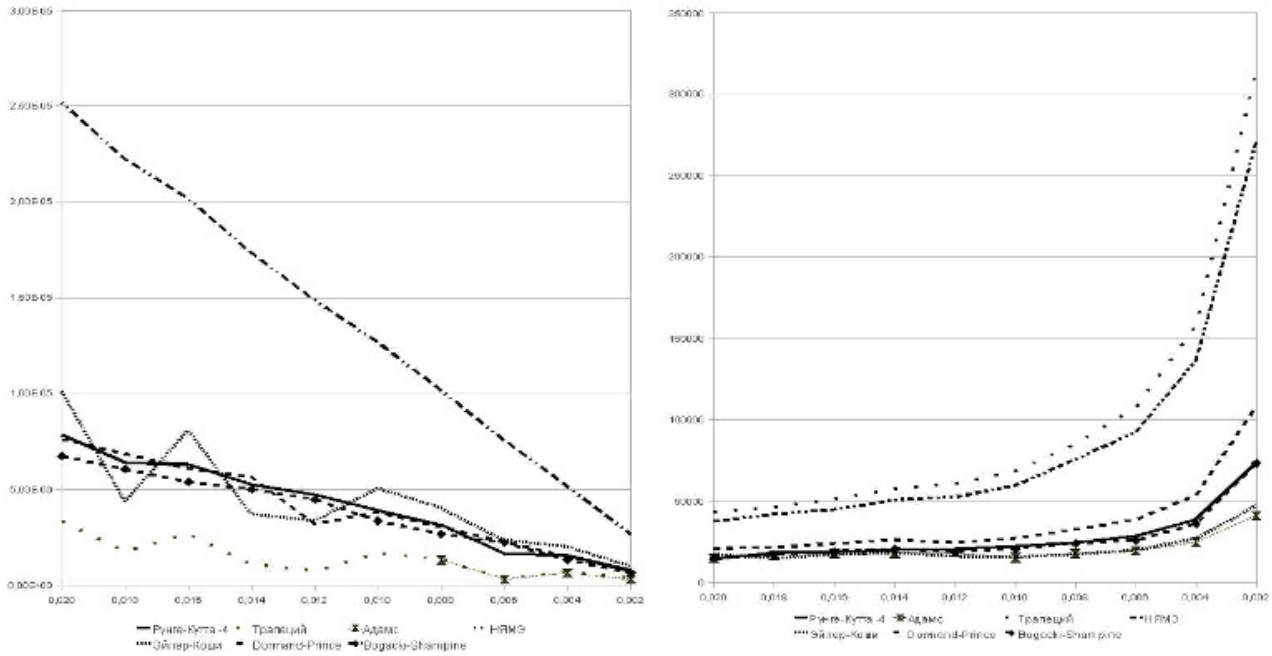
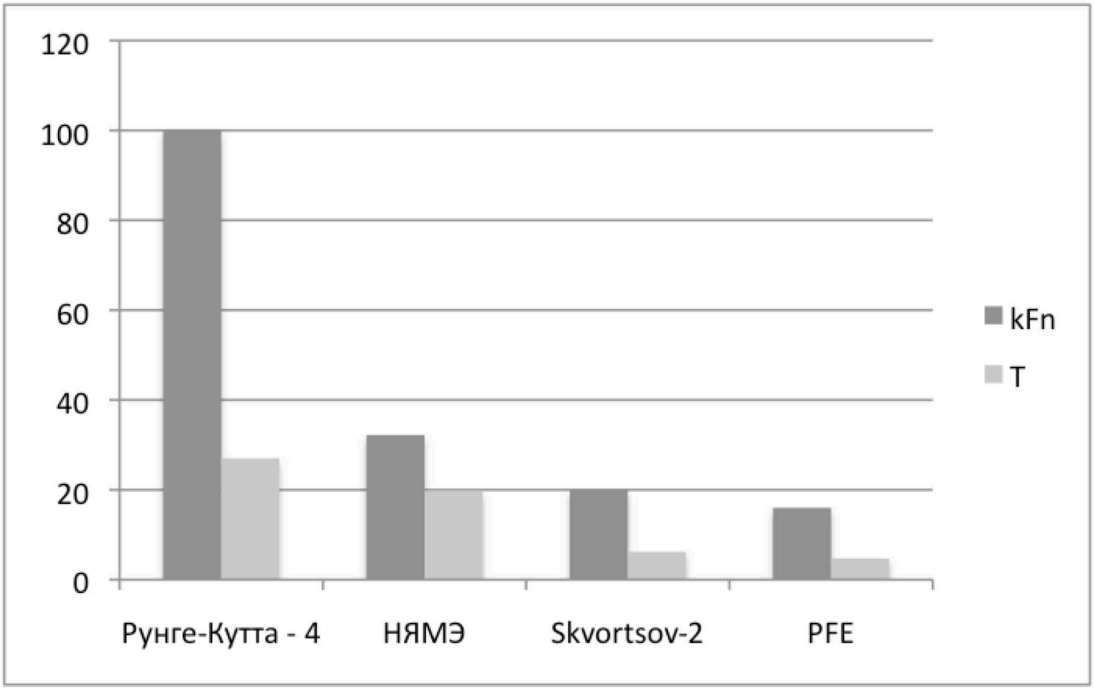


Рисунок 4.1. Решение системы для N=1 различными методами

не будем проводить подробного анализа жесткости системы с вычислением собственных значений, так как, как уже говорилось, классическое определение жесткости использует неопределенное понятие «малости» собственных чисел и, соответственно, не позволяет получить каких-либо строгих результатов. Вместо этого попробуем подтвердить наше предположение о жесткости системы, использовав для ее решения несколько методов с более широкой областью устойчивости, чем у метода Рунге–Кутты 4-го порядка. Как известно, наиболее распространенный способ решения жестких систем - использование неявных A-устойчивых методов, недостатки которых также хорошо известны – сложность в реализации и вычислительная неэффективность. Для нашего вычислительного эксперимента будем использовать простейший устойчивый неявный метод - неявный метод Эйлера, поскольку он прост в реализации, обладает свойствами A- и L-устойчивости и требует на каждый шаг не больше вычислений, чем любой другой неявный метод.

Таблица 4.1. Решение системы для N=20.

Метод	h	F_n	T
Рунге-Кутта - 4	0.04	100000	270
НЯМЭ	4	32186	197
Skvortsov-2	0.2	20000	62
PFE	0.25	16000	47



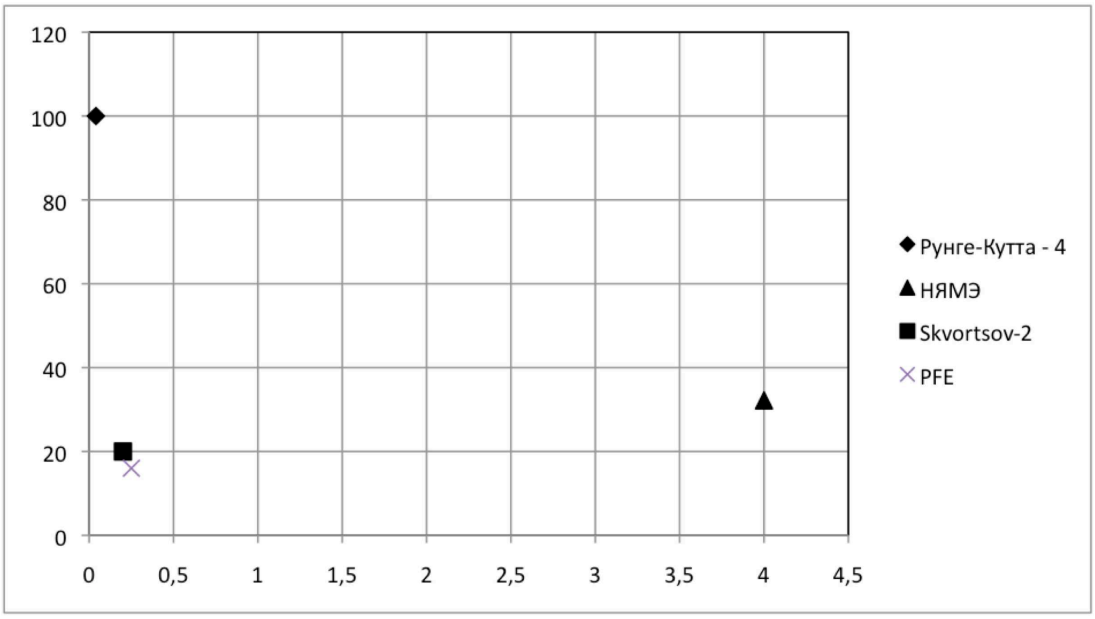


Рисунок 4.2. Сравнительная эффективность методов

Результаты численного эксперимента приведены в таблице 4.1(более подробно см. [11]), где h – максимальный шаг, обеспечивающий точность; F_n– количество вычислений правой части; T – время вычислений в миллисекундах на некоторой тестовой машине (Intel Core i5 750@~2.66 GHz). Нужно отметить несколько особенностей, касающихся использования неявного метода Эйлера. Во-первых, для решения системы алгебраических уравнений использовался упрощенный метод Ньютона, для решения линейной системы – метод LUP–разложения. Матрица Якоби системы вычислялась автоматически на основе формулы центральной разности:

$$y^{'}(t) \approx \frac{y(t + \triangle t) - y(t - \triangle t)}{2\triangle t}.$$

Во-вторых, ошибка в конечной точке интервала для неявного метода Эйлера оставалась в заданных пределах и при размере шага больше 4с, однако использование таких крупных шагов не оправдано практически, так как в таком случае есть риск пропустить особенности поведения целевого значения по масштабу, меньшие, чем размер шага. Поэтому максимальный размер шага был достаточно произвольно ограничен 4 с. Несмотря на это, можно видеть, что неявный метод Эйлера все равно требует меньше времени на вычисления, чем метод Рунге–Кутты, что является практическим свидетельством жесткости системы. Наиболее эффективными же оказываются явные жесткие методы, разница в производительности между которыми относительно невелика. Проекционный метод является наиболее эффективным, однако метод Скворцова при незначительно меньшей эффективности обеспечивает больший порядок точности, что, учитывая инерционность изменения параметра x, на наш взгляд для данной системы не является важным. Поэтому будем считать что для рассматриваемой системы наиболее подходящими являются проекционные методы. Таким образом мы можем считать, что данная система обладает необходимыми особенностями, чтобы продемонстрировать на ее примере эффективность проекционных методов.

4.3. Выводы

В данной главе было выполнено исследование численного решения системы 3.2 с различными значениями параметров несколькими численными методами и получены следующие результаты:

- система 3.2 с одним источником является нежесткой в смысле практического определения жесткости, данного в параграфе 1.1.3;
- система с 20 источниками является жесткой, так как специализированные методы для решения жестких ОДУ имеют для данной системы преимущество в производительности;
- явные методы с расширенной областью устойчивости имеют существенно большую производительность, чем неявные устойчивые методы.

Таким образом, использование явных методов с расширенной областью устойчивости, в частности проекционных методов, для данной задачи обосновано и имеет существенное преимущество в производительности.

5. Применение GPGPU для интегрирования системы

5.1. Мотивация и условия применимости

Рассмотрим вслед за [15] реализацию решения системы 3.2 для N порядка нескольких тысяч источников. Для простоты, примем что количество источников является степенью двойки (далее будет показано, чем вызвано такое требование). На решении реальной задачи такое упрощение не скажется, так как мы всегда можем дополнить систему виртуальными источниками, не влияющими на решение.

5.2. Стратегия распараллеливания

5.2.1. Распределение вычислений по потокам

В данной работе мы акцентируем внимание на решении жестких систем явными методами, для которых основным является распараллеливание вычисления правой части системы. Наиболее очевидная стратегия распараллеливания любой системы - выделить по одному потоку на каждое уравнение. Однако от такого решения было принято отказаться по ряду причин:

- разные уравнения требуют разного количества вычислительных операций, соответственно загрузка вычислительных устройств будет неравномерной;
- вычисления, выполняемые каждым потоком, имеют небольшой объем, в результате чего на производительность большее влияние начинают оказывать накладные расходы на вызов функций, а не вычисления.

В связи с этим будем использовать по одному потоку на каждый источник, т. е. каждый поток будет отвечать за решение трех уравнений. Таким образом мы добиваемся более равномерной загрузки вычислительных устройств.

Что касается конфигурации блоков, выберем одномерную последовательность с 128 потоками в блоке. Число 128 выбрано исходя из следующих соображений:

- является степенью двойки, что упрощает реализацию редукции;

– содержит 4 warp, что обеспечивает загрузку исполнительных модуле GPU (warp является минимальной единицей, которой оперирует планировщик потоков графического процессора). Таким образом, наличие 4 warp на блок и, соответственно, на исполнительное устройство позволяет маскировать время выполнения длительных асинхронных операций, например обращений к памяти;
– для рассматриваемого количества уравнений обеспечивает достаточное количество блоков для загрузки всех исполнительных устройств современных GPU.

5.2.2. Преобразования математической модели

Если запрограммировать интегрирование системы 3.2 непосредственно по формулам 3.1, производительность полученной программы будет значительно меньше ее производительности на CPU. Чтобы получить существенное ускорение, нужно немного преобразовать модель для лучшего распараллеливания.

Рассмотрим формулу 3.1:

$$Y_i = \frac{\frac{\psi_i}{r_i} x_i (\sum \frac{R}{r_j} - 1) - \frac{R}{r_i} \sum \frac{\psi_j}{r_j} x_j}{1 - \sum \frac{R}{r_j}}.$$

Во-первых, можно заметить, что коэффициенты ψ, R, r не меняются на протяжении решения, поэтому можно предварительно рассчитать значения $\sum \frac{R}{r_j}$ и подобные. На CPU такая оптимизация также возможна, однако не даст значительного прироста эффективности, тогда как на GPU, где время обращения к глобальной памяти на несколько порядков превышает время исполнения арифметических операций, эффект от такой оптимизации будет значителен. Немаловажно также то, что вычисление $\sum \frac{R}{r_j}$ требует обращения к памяти, обрабатываемой другими потоками, то есть фактически операции редукции (хотя и упрощенной, так как значение не меняется). В выражении $\frac{R}{r_i} \sum \frac{\psi_j}{r_j} x_j$ значение x_j меняется на каждом шаге, так что для вычислений потребуется полноценная редукция, что сложно организовать в середине функции на графическом процессоре. Поэтому выделим специальную область в глобальной памяти, где будет храниться результат редукции и будем обновлять его при каждом вычислении правой части. Задачу можно немного упро-

стить, потребовав, чтобы для всех источников $\psi_i = \psi$, в таком случае уравнение для вычисления Y_i можно переписать следующим образом:

$$Y_i = \frac{(S_{Rr} - 1)\Psi_i x_i - \frac{R}{r_i}\Psi_i S_x}{1 - S_{Rr}},$$

где S_{Rr}– предварительно вычисленное значение $\sum \frac{R}{r_j}$; Ψ_i – предварительно вычисленное значение $\frac{\psi}{r_i}$; S_x – результат редукции $\sum x_j$.

Таким образом, вычисление правой части системы уравнений примет следующий вид:

1. CPU:
   - 1.1. Предвычислить константы S_{Rr}, Ψ_i.
   - 1.2. Предвычислить S_xна основе начальных значений x.
2. GPU:
   - 2.1. Вычислить правую часть, используя значения, полученные на шагах 1–2.
   - 2.2. Обновить S_x.
   - 2.3. Последующие вычисления правой части начинать с шага 2.1.

Таким образом, в соответствии с принципами, описанными в главе 2.2, нам удалось организовать вычисления таким образом, что операция редукции вынесена в конец вычисления, что значительно упрощает ее реализацию на графическом процессоре, так как глобальная синхронизация потоков на нем возможна только при завершении функции ядра.

5.3. Результаты численного эксперимента

Для того чтобы продемонстрировать преимущество интегрирования системы 3.2 на GPU, приведем результаты численного эксперимента. Рассмотрим интегрирование систем с 2048, 4096 и 8192 источниками, что соответствует 6144, 12288 и 24576 уравнениям. Сравнивать будем три реализации: однопоточную реализацию на CPU, многопоточную реализацию на CPU и реализацию на GPU. В качестве тестового стенда использовался персональный компьютер с процессором Intel Core i5 750@2.66 GHz, имеющим 4 вычислительных ядра, а также видеокартой NVIDIA GeForce GTX 580, имеющей 512 потоковых процессоров.

Таблица 5.1. Время интегрирования системы методом Эйлера

N	τ_1, CPU-ST	τ_2, CPU-MT	$\frac{\tau_1}{\tau_2}$	τ_3,GPU	$\frac{\tau_2}{\tau_3}$
2048	4,64	2,04	2,27	0,14	14,57
4096	9,29	3,22	2,89	0,14	23
8192	18,59	6,12	3,04	0,19	32,21

Таблица 5.2. Время интегрирования системы проекционным методом

N	τ_1, CPU-ST	τ_2, CPU-MT	$\frac{\tau_1}{\tau_2}$	τ_3, GPU	$\frac{\tau_2}{\tau_3}$
2048	19,14	8,67	2,21	0,54	16,06
4096	38,96	13,9	2,8	0,61	22,79
8192	77,85	25,46	3,05	0,75	33,95

В таблицах 5.1и 5.2 приведены результаты численного эксперимента для систем вида 3.2, то есть включающих только источники, но не потребителей. Время вычислений с использованием проекционного метода больше времени вычислений методом Эйлера, так как мы использовали одинаковое значение шага по времени для обоих методов, для проекционного метода он мог быть увеличен как минимум в несколько раз, однако для рассматриваемого вопроса сравнения производительности реализаций на центральном и графическом процессорах выбор длины шага не играет роли. Как можно видеть, для рассмотренных конфигураций системы уравнений использование GPU дает значительный прирост производительности (до 34 раз). Интересно отметить, что при увеличении числа уравнений вдвое, время вычислений на CPU также возрастает примерно вдвое, тогда как для графического процессора время выполнения возрастает в гораздо меньшей степени. Это свидетельствует о том, что, несмотря на проведенные оптимизации, программа все равно не задействует все ресурсы графического процессора.

Далее рассмотрим решение варианта системы с небольшим количеством источников и большим количеством потребителей. Из соображений простоты реализации и сравнения результатов с результатами предыдущего численного эксперимента сохраним общее количество уравнений тем же, но установим, что только 16 устройств являются источниками, а остальные - потребителями. В таком случае, очевидно, количество вычислений сократится (поскольку нет необходимости считать ρ для большинства объектов). Время вычислений на CPU, как и следовало ожидать, сокращается (табл. 5.3 и 5.4). Однако время вычислений на GPU остается практически неизменным, что связано с тем фактом, что с уменьшением времени вычислений каждым ядром, возрастает

относительная доля накладных расходов на вызовы графического процессора в общем времени вычислений. Таким образом, мы видим, что при большом числе потребителей реализация на GPU становится относительно менее эффективной, однако в абсолютных значениях все равно превосходит реализацию на центральном процессоре в 8 и более раз, то есть вне зависимости от соотношения потребителей и источников, использование графических процессоров для решения систем подобного типа имеет существенное преимущество по производительности перед использованием центрального процессора.

Таблица 5.3. Время интегрирования системы методом Эйлера

N	τ_1, CPU-ST	τ_2, CPU-MT	$\frac{\tau_1}{\tau_2}$	τ_3, GPU	$\frac{\tau_2}{\tau_3}$
2048	2,59	1,17	2,22	0,14	8,35
4096	4,52	1,8	2,51	0,14	12,85
8192	9,28	3,01	3,08	0,19	15,84

Таблица 5.4. Время интегрирования системы проекционным методом

N	τ_1, CPU-ST	τ_2, CPU-MT	$\frac{\tau_1}{\tau_2}$	τ_3, GPU	$\frac{\tau_2}{\tau_3}$
2048	11,9	5,31	2,24	0,56	9,48
4096	23,81	9,05	2,63	0,61	14,84
8192	46,93	15,05	3,12	0,76	19,79

5.4. Выводы

В данной главе было показано, что использование графических процессоров общего назначения в качестве аппаратной платформы для решения систем 3.2 и 3.3 приводит к значительному увеличению производительности решения. Для системы 3.3 относительный прирост производительности по сравнению с решением на центральном процессоре ниже, однако и в этом случае он остается достаточно существенным. Таким образом, поставленная задача создания высокопроизводительного программного комплекса для решения жестких систем ОДУ решена.

6. Библиотека программ для решения систем ОДУ

6.1. Общая информация

В процессе работы над данным диссертационным исследованием была разработана библиотека программ, реализующая многие из описанных методов интегрирования, а также связанные с ними численные методы, например различные модификации метода Ньютона для решения систем нелинейных алгебраических уравнений, а также различные алгоритмы решения систем линейных алгебраических уравнений. Цели разработки библиотеки были следующие:

1. *Современный, простой в использовании интерфейс.* Многие профессиональные наборы библиотек в силу своей универсальности требуют большого количества подготовительного кода, что допустимо непосредственно при моделировании, но чрезвычайно осложняет практическое исследование самих методов решения. Более того, интерфейс многих библиотек сложился задолго до того как программирование сформировалось как инженерная дисциплина, что усложняет их использование.
2. *Высокая производительность.* Многие современные средства решения систем ОДУ обеспечивают посредственную производительность, например некоторые подпрограммы систем компьютерной математики (MATLAB и в большей степени GNU Octave).
3. *Высокий уровень абстракции и как следствие - переносимость.* Требуется возможность легко добавлять поддержку специализированных вычислительных устройств, таких как графические процессоры. При этом желательно сократить объем дублирования кода, что ведет к уменьшению количества ошибок.

Библиотека реализована на языке C++, поскольку этот язык может обеспечить как высокую производительность, так и достаточно высокий уровень абстракции.

6.2. Структура

Высокий уровень абстракции достигается следующими средствами. Во-первых, алгоритм вычисления одного шага по времени отделен от алгоритма решения системы в целом. Шаг по времени реализуется конкретным методом интегрирования (например методом Эйлера или Рунге-Кутты), тогда как алгоритм решения системы в целом определяет стратегию и последовательность шагов, управляет размером шага по времени и может выбирать конкретный метод интегрирования. Другими словами, каждый метод интегрирования представлен классом (например, Rk4Step для метода Рунге-Кутты 4-го порядка), который реализует один шаг этого метода. Данный класс(или несколько классов) используется функцией интегрирования, находящей непосредственно искомое решение системы. Таким образом, имея k методов и m «решателей», мы получаем km различных способов решения системы (рисунок 6.1).

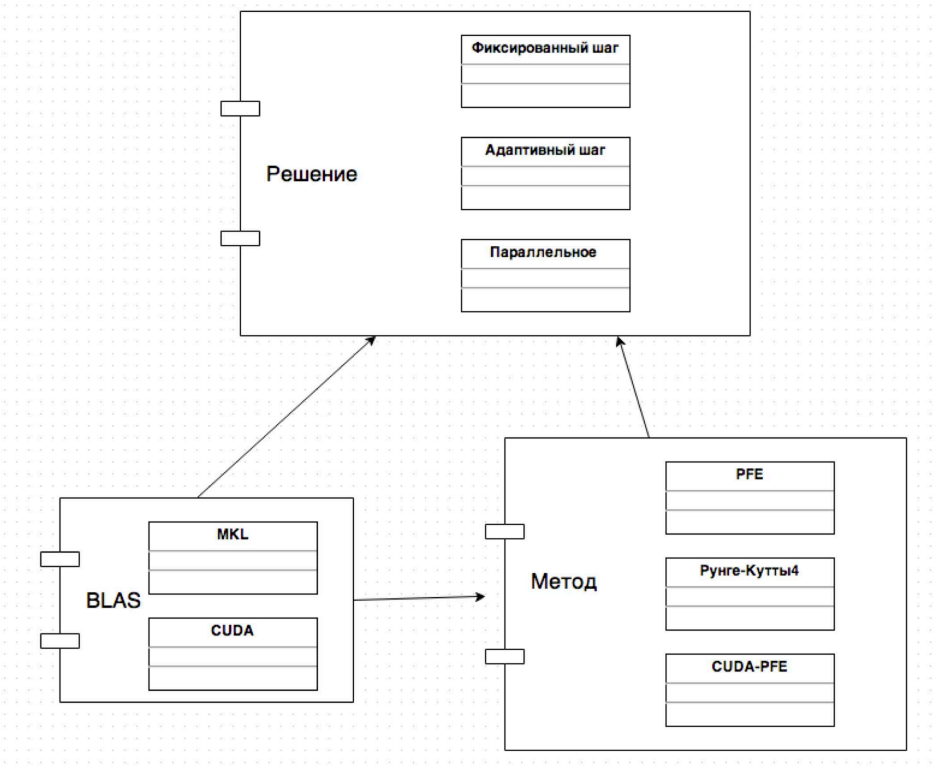


Рисунок 6.1. Общая структура библиотеки программ

Внутренняя реализация библиотеки использует для непосредственно вычислений набор BLAS-подобных функций, реализующих операции векторной алгебры. эти функции объединены в классы, которыми параметризуются все вычисления. Существует несколько классов, реализующих эти функции, один

из них использует реализацию BLAS из библиотеки Intel Math Kernel Library, другой использует CuBLAS. Пользователь может использовать свою реализацию, передав ее в функцию решения. Таким образом мы абстрагируем конкретный алгоритм решения системы ОДУ от способа вычислений конкретных значений переменных, что позволяет использовать один и тот же код для решения задачи, например, на CPU и GPU, но при этом не мешает в случае необходимости реализовывать, например, всю цепочку решения на GPU (это, например, сделано для описанного в главе 5 решения).

6.3. Реализованные алгоритмы

В библиотеке программ реализованы следующие методы решения ОДУ.

- явные методы Рунге-Кутты:
  - метод Рунге-Кутты 4-го порядка;
  - метод Дормана-Принса;
  - метод Богацки-Шампайна;
- явные методы Эйлера:
  - явный метод Эйлера;
  - метод Хойна;
- неявные методы Эйлера:
  - неявный метод Эйлера;
  - метод трапеций;
  - метод Симпсона;
  - метод Тика;
  - метод «трех восьмых»;
- диагонально-неявные методу Рунге-Кутты:
  - метод 3-го порядка;
  - метод NT1;
  - L-стабильный метод;
- неявные методы Рунге-Кутты:
  - метод Радо 3-го порядка;
  - метод Гаусса 4-го порядка;
  - метод Лобатто 6-го порядка;

- явные методы с расширенной областью устойчивости:
  - методы Рунге-Кутты-Чебышёва;
  - методы Скворцова 1-го, 2-го, 3-го порядков;
  - проекционный метод Эйлера.

Помимо этого, реализованы следующие вспомогательные алгоритмы, которые могут быть использованы самостоятельно:

- решение систем нелинейных алгебраических уравнений:
  - модифицированный метод Ньютона;
  - упрощенный метод Ньютона;
- решение СЛАУ:
  - прямые методы:
    - LU- разложение;
    - LUP - разложение;
  - итерационные методы:
    - Bicg-Stab;
    - метод сопряженных градиентов;
    - CGS;
    - метод Чебышёва.

Реализованы следующие варианты BLAS:

- RefBlas - «наивная» реализация, предназначена в основном для отладки, функции работы с векторами реализованы простыми циклами без каких-либо оптимизаций;
- MKLBlas - обертка над реализацией BLAS из Intel Math Kernel Library [48];
- PMKLBlas - дополнительно распаралеленные при помощи Intel TBB [49] функции Math Kernel Library;
- CudaBlas - использует CUDA.

6.4. Пример использования и тестирование

Для построения юнит-тестов библиотеки используется система google-test [41]. Каждый метод интегрирования тестируется на двух простых системах уравнений. Из кода тестов же можно взять пример использования:

```
inline void sin_eq(double t, const double* x, double* F) {
        F[0] = 3*sin(4*t);
}

solve_fixedstep
<double,                    //тип данных элемента вектора
RefBlas,                    //BLAS
Rk4Step             //Метод интегрирования
>
(       1,                  //размерность системы
        0.0f,       //начало интервала интегрирования
        dend,       //конец интервала интегрирования
        step,       //размер шага по времени
        sin_eq,     //функция вычисления правой части
        init.data(), //вектор начальных значений
        result.data() //результат
);
```

7. Основные результаты работы

В данной работе были получены следующие научные результаты:

- Предложено семейство систем ОДУ, позволяющее моделировать широкий класс различных технических систем, обладающих рядом общих свойств и сводимых друг к другу с использованием принципа динамических аналогий.
- Показано, что свойства системы дифференциальных уравнений определяются ее размерностью. Так, система уравнений для одного источника является нежесткой, тогда как система уравнений для нескольких источников является жесткой. Следовательно, распространенные явные методы (Эйлера, Рунге-Кутты и т. д.) для подобных систем неэффективны и требуется использование одного из специализированных методов интегрирования.
- Показано, что проекционный метод Эйлера эффективно решает рассмотренную систему, существенно превосходя при этом по производительности неявные A-устойчивые методы, которые являются стандартными для интегрирования жестких систем.
- Предложен способ программной реализации интегрирования рассмотренной системы уравнений на специализированном вычислительном устройстве - графическом процессоре общего назначения.
- Проведенные численные эксперименты показали существенное (в 10 и более раз) преимущество по производительности предложенной программной реализации на графическом процессоре перед реализацией на центральном процессоре и подтвердили высокую эффективность использования графических процессоров для моделирования технических систем высокой размерности.

Список литературы

1. Агафонов, С.А, Дифференциальные уравнения / С.А. Агафонов, А.Д. Герман, Т.В. Муратова. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2004. – 352 с.

2. Амосов, А.А. Вычислительные методы для инженеров: учеб. пособие / А.А. Амосов, Ю.А. Дубянский, Н.В. Копченов. — М.: Высш. шк., 1994. — 544 с.

3. Березин И. С., Жидков Н. П., Методы вычислений / И.С. Березин, Н.П. Жидков.– 3 изд. – М., 1966. — 632 с.

4. Кокин, В.М. Моделирование систем: учеб. пособие / В. М. Кокин. — Иваново: Б.и., 2002.—116 с.

5. Колебания и бегущие волны в химических системах: пер. с англ./под ред. Р. Филда, М. Бургер. — М.: Мир, 1988. – 720 с., ил.

6. Мудров, А.Е., Численные методы для ПЭВМ на языках Бейсик, Фортран и Паскаль /А.Е. Мудров. – Томск: МП Раско, 1991.– 272 с.

7. Понтрягин, Л.С. Обыкновенные дифференциальные уравнения / Л.С. Понтрягин. – 4-е изд. - М.: Наука. - 1974 .– 331 с.

8. Скворцов Л.М. Простые явные методы численного решения жестких обыкновенных дифференциальных уравнений / Л.М. Скворцов // Вычислительные методы и программирование. – 2008. – № 9. – С. 154–162.

9. Скворцов, Л.М. Явные многошаговые методы численного решения жестких обыкновенных дифференциальных уравнений / Л.М. Скворцов // Вычислительные методы и программирование. – 2008. – № 9. – С. 409–418.

10. Советов, Б.Я. Моделирование систем: учеб. пособие для вузов / Б.Я. Советов, С.Я. Яковлев. — 5-е изд., стер. - М.: Высш.шк.,2007. – 2007. – 343 с.

11. Чадов, С.Н. Интегрирование жесткой модели электромеханической системы явными методами/ С.Н. Чадов // Вестн. ИГЭУ. – 2012. – № 2. – С. 73–75.

12. Чадов С.Н. Некоторые вопросы численного моделирования динамических систем / ГОУВПО «Ивановский государственный энергетический университет им. В.И. Ленина». – Иваново, 2010. – 120 с.

13. Чадов, С.Н. О решении СЛАУ методом итераций Чебышёва на графических процессорах / С.Н. Чадов // Вестн. ИГЭУ. – 2010. – № 3. – С. 76–78.

14. Чадов, С.Н. Реализация алгоритма решения несимметричных систем линейных уравнений на графических процессорах / С.Н. Чадов // Вычислительные методы и программирование. – 2009. – №10. – С. 321–326.

15. Чадов, С.Н. Реализация проекционного явного метода решения жесткой системы ОДУ на графическом процессоре общего назначения / С.Н. Чадов // Вестн. ИГЭУ. – 2013. – № 4. – С. 79–82.

16. Чадов, С.Н. Исследование производительности численного алгоритма решения жестких систем дифференциальных уравнений / С.Н. Чадов // Вестн. ИГЭУ. – 2007. – № 4. – С. 26–29.

17. Чадов, С.Н. Численное исследование модели энергетической системы / С.Н. Чадов // Вестн. ИГЭУ. – 2009. – № 4. – С. 49-52.

18. Чадов, С.Н. Экспериментальное исследование явных методов решения обыкновенных дифференциальных уравнений с расширенной областью устойчивости / С.Н. Чадов // Вестн. ИГЭУ. – 2009. – № 4. – С. 52-54.

19. Эйлер, Л. Интегральное исчисление, т. 1 / Л. Эйлер.– М.: Гостехиздат, 1956. – 415 с.

20. Abdulle, A. Fourth Order Chebyshev Methods with Recurrence Relation / A. Abdulle // SIAM J. Sci. Comput. – Vol. 23, № 6. – P. 2041–2054.

21. Ascher, U. M.. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations / U.M. Ascher, S.J. Ruuth, R.J. Spiteri // Applied Numerical Mathematics. – 1997. – 25(2). – P. 151–167.

22. Barrett, R. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods / R. Barrett, M. Berry, T.F. Chan, et al. – 2nd Edition. – Philadelphia: SIAM, 1994.

23. Benzi, M. A comparative study of sparse approximate inverse preconditioners / M. Benzi, M. Tůma // Applied Numerical Mathematics. – 1999. – 30. – P. 305–340.

24. Benzi, M. Preconditioning Techniques for Large Linear Systems: A Survey / M. Benzi // Journal of Computational Physics. – 2002. – 182. – P. 418–477.

25. Bogacki, P. A 3(2) pair of Runge–Kutta formulas / P. Bogacky, L.F. Shampine //Applied Mathematics Letters. – 1989. – 2 (4). – P. 321–325.

26. Buatois, L. Concurrent number cruncher: a GPU implementation of a general sparse linear solver / L. Buatois, G. Caumon, B. Lévy // International Journal of Parallel, Emergent and Distributed Systems. – 2009. – Vol. 24, № 3. – P. 205–223.

27. Butcher, J. C., Numerical Methods for Ordinary Differential Equations / J.C. Butcher. – John Wiley, 2008. – 482 P.

28. Butcher, J.C., On Runge-Kutta Processes of high order/ J.C. Butcher // J. Austral. Math. Soc. – 1964. – vol. IV, Part 2. – P. 179–194.

29. Butcher J. Runge-Kutta methods. [Электронный ресурс]. – URL: http://www.scholarpedia.org/article/Runge-Kutta_methods.

30. Chen, Y. An Accelerated Block-Parallel Newton Method via Overlapped Partitioning / Y. Chen, ed. T.J. Barth, M. Griebel, D.E. Keyes, et al // Domain Decomposition Methods in Science and Engineering. – Berlin: Springer Berlin Heidelberg, 2005. – P. 547–554.

31. Cormen, T. Introduction to algorithms / T. Cormen, C. Leiserson, R.Rivest. – 2nd ed.– MIT Press, 2010. – 1180 P.

32. Curtiss C. F. Integration of Stiff Equations. / C.F. Curtiss, J.O. Hirschfelder // Proc Natl. Acad. Sci. USA. – 1952 – 38(3). – P. 235–243.

33. Dormand, J. R. A family of embedded Runge-Kutta formula / J.R. Dormand, P.J. Prince // Journal of Computational and Applied Mathematics. – 1980. – 6 (1). – P. 19–26.

34. Ehle, B. L. On Padé approximations to the exponential function and A-stable methods for the numerical solution of initial value problems: a thesis ... doctor of philosophy / B.L. Ehle. – Waterloo, 1969.

35. Ekeland, K. Stiffness Detection and Estimation of Dominant Spectrum with Explicit Runge-Kutta Methods / K. Ekeland, B. Owren, E. Øines // ACM Trans. On Math. Soft. – 1998. – Vol 24, № 4. – P. 368–382.

36. Field, R. J. Oregonator. [Электронный ресурс]. URL: http://www.scholarpedia.org/article/Oregonator

37. Gear, C.W. Numerical Initial Value Problems in Ordinary Differential Equations. / C.W. Gear. – NJ: Prentice-Hall, Englewood Cliffs. – 1971.

38. Gear, C. W. Projective methods for stiff differential equations: Problems with gaps in their eigenvalue spectrum / C. W. Gear, I. G. Kevrekidis // SIAM J. Sci. Comput. – 203. – 24 (4). – P. 1091–1106.

39. Gear C. W. Telescopic projective methods for parabolic differential equations/ C.W. Gear // J. Comput. Phys. – 2003 .– 187 (1). – P. 95–109.

40. Gear, C.W. The potential for parallelism in ordinary differential equations / C.W. Gear // Tech. Rep. UIUCDCS-R-86-1246. – Urbana: University of Illinois, 1986.

41. Google test documentation [Электронный ресурс]. – URL: https://code.google.com/p/googletest/wiki/Primer

42. Hairer, E. Solving ordinary differential equations II: Stiff and differential-algebraic problems. / E.Hairer, G. Wanner. — 2nd ed. — Berlin: Springer-Verlag, 1996.

43. Harris M., Optimizing parallel reduction in CUDA: Tech. Report / NVIDIA, 2007.

44. Hindmarsh, A. C. User Documentation for cvode v2.4.0. / A. C. Hindmarsh, R. Serban. – Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2006.

45. Kanamaru T. Van der Pol oscillator [Электронный ресурс]. – URL: http://www.scholarpedia.org/article/Van_der_Pol_oscillator

46. Lambert, J. D. Numerical Methods for Ordinary Differential Systems. / J.D. Lambert. – New York: Wiley, 1992.

47. Lee, S. L.. Second-order accurate projective integrators for multiscale problems / S.L. Lee, C.W. Gear //Journal of Computational and Applied Mathematics. – 2007 . – 201(1). – P. 258-274.

48. Intel MKL documentation [Электронный ресурс]. URL: http://software.intel.com/en-us/intel-mkl

49. Intel TBB documentation [Электронный ресурс]. URL: http://threadingbuildingblocks.org/

50. Kennedy, C. A. Additive Runge-Kutta schemes for convection-diffusion reaction equations / C.A Kennedy, M.H. Carpenter // Appl. Numer. Math. – 2003. – № 44. – P. 139–181.

51. Medovikov A.A. Third Order Explicit Method for the Stiff Ordinary Differential Equations / A.A Medovikov // Numerical analysis and Its Applications, First International Workshop. – WNAA, 1996

52. NVIDIA. CUDA Programming Guide, 2.1 edition, 2009. [Электронный ресурс]. – URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

53. Ortega, J.M. Introduction to parallel and vector solution of linear systems / J.M. Ortega. – N.Y.: Plenum Press, 1988. – 318 P.

54. Østergaard, E. Documentation for the SDIRK C++ solver, IMM, ITU, 1998.

55. Prigogine, I. Symmetry Breaking Instabilities in Dissipative Systems / I. Prigogine, R. Lefever // J. Chem. Phys. – 1968. – 48, 1695.

56. Shampine L.F., Lipschitz constants and robust ODE codes. / L.F. Shampine, J.T. Oden (Ed.) // Computational Methods in Nonlinear Mechanics. – New York: North-Holland, 1980. – P. 427–449.

57. Shampine, L.F. Numerical Solution of Ordinary Differential Equations / L.F. Shampine. – NY: Chapman & Hall, 1994.

58. Spiteri R.J., Stiffness Detection in Initial-Value ODEs. Dept. of Mathematics and Statistics Faculty of Computer Science Dalhousie University. [Электронный ресурс]. – URL: http://www.math.mcgill.ca/humphries/research/caims/spiteri04

59. Van der Houwen, P.J. On the Internal Stability of Exlplicit m-stage Runge-Kutta methods for large m-values / P.J van der Houwen // Z. Angew. Math. Mech. – 1980. – № 60. – P. 479–485.

60. Van der Houwen, P. J. Parallel iteration of high-order runge-kutta methods with stepsize control / P.J. van der Houwen, B.P. Sommiejer //Journal of Computational and Applied Mathematics. – 1990. – № 29. – P. 111–129.

61. Verwer, J.G. An Implicit-Explicit Runge-Kutta-Chebyshev Scheme for Diffusion-Reaction Equations / J.G. Verwer, B.P. Sommeijer // Report MAS-R0305. – Amsterdam: Stichting Centrum voor Wiskunde en Informatica, 2003.

62. Volkov, V. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. / V. Volkov, J.W. Demmel // EECS Department, University of California, Berkeley Tech. rep. – 2008.

63. Wanner, G. Solving Ordinary Differential Equations / G. Wanner, E. Hairer, S.P. Nørsett. – New York: Springer-Verlag New York, Inc., 1993.

64. White, J. Waveform relaxation: theory and practice // J. White, A Sangiovanni-Vincentelli, F. Odeh, et al. // Trans. Soc. Comput. Sim. – 1985. – No 2. – P. 95–133.

65. Kutta, W. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen / W. Kutta // Zeitschr. für Math. u. Phys. – vol 46. – P. 435-453.

66. Nielsen, H. B, Thomsen P.G, Hæfte 66 - Numeriske Metoder for Sædvanlige differentialligninger, Numerisk Institut, DTH, 1993.

Приложение

В этом приложении приведены фрагменты исходного кода программ решения систем ОДУ.

Исходный код операции редукции на GPU.

```
__device__
void reduce_device(
  real_t* reduced_ptr,     //указатель на целевую область памяти
  real_t* reduce_buffer,   //результат, должен содержать достаточно
     памяти для хранения результта редукции каждого блока
  int reduce_size      //размер блока reduced_ptr
  ){

  //выполнить редукцию в каждом блоке отдельно
  //соответсвует первому этапу двухэтапного алгоритма
  reduce2<real_t,THREADS_PER_BLOCK,false,false>(reduced_ptr,
     reduce_buffer,reduce_size);

  const unsigned int tid = threadIdx.x;
  //разделяемая переменная, устанавливается в true, если текущий
     блок последний
  __shared__ bool amLast;

  //барьер памяти
  //необходим для того, чтобы все изменения в глобальной памяти были
      доступны всем потокам
  //если барьер убрать, возможна такая ситуация, что текущему потоку
      будут недоступны измеения, сделанные потоком из другого warp
  __threadfence();

  //нулевой поток каждого блока увеличивает счетчик
  if( tid==0 ) {
    //важно, чтобы увеличение счетчика было атомарным, в противном
       случае получаем race condition
    unsigned int ticket = atomicInc(&retirementCount, gridDim.x);

    //определяем является ли текущий блок последним
    amLast = (ticket == gridDim.x - 1);
  }
```

```
    //синхронизация потоков внутри блока, нужна для того все потоки
        блока дождались заполнения значения amLast
    __syncthreads();

    // Последний блок выполняет второй этап редукции
    if(amLast) {
      reduce2<real_t,THREADS_PER_BLOCK,true,false>(reduce_buffer,
          reduce_buffer,gridDim.x);
      if( tid==0 ){
        // сбрасываем счетчик, чтобы следующий вызов функции дал
            верные результаты
        retirementCount = 0;
      }
    }
}

template < class T,         //тип данных, в нашем случае всегда float
unsigned int blockSize,   // размер блока
bool last          //true,если это второй этап редукции
>
__device__  void reduce2(
  T *g_idata,
  T *g_odata,
  unsigned int n
){
  //объявляем массив в разделяемой памяти
    __shared__ T __smem[THREAD_BLOCK_SIZE];
  T *sdata = &__smem[0];

  // загружаем данные в разделяемую память
  unsigned int tid = threadIdx.x;
  unsigned int i = last? tid : compute_thread_index();
  sdata[tid] = (i < n) ? g_idata[i] : 0;

  //синхронизация внутри блока.
  //После этой точки все данные для этого блока уже загружены в
      разделяемую память
  __syncthreads();

  // выполняем редукцию в разделяемой памяти
  //цикл намеренно развернут для более быстрого выполнения
```

```
  // если мы используем 128 потоков на блок, первые два условия
     отбрасываются на этапе компиляции
  if(blockSize >= 512) { if(tid < 256) { sdata[tid] += sdata[tid
     +256]; } __syncthreads(); }
  if(blockSize >= 256) { if(tid < 128) { sdata[tid] += sdata[tid
     +128]; } __syncthreads(); }
  if(blockSize >= 128) { if(tid <  64) { sdata[tid] += sdata[tid+
     64]; } __syncthreads(); }

  //оставшиеся элементы можно сложить потоками одного warp
  //потоки одного warp выполняются по принципу  SIMD
  //соответсвенно синхронизация между ними не требуется
  if(tid<32){
    if (blockSize >= 64) sdata[tid] += sdata[tid+32];
    if (blockSize >= 32) sdata[tid] += sdata[tid+16];
    if (blockSize >= 16) sdata[tid] += sdata[tid+ 8];
    if (blockSize >=  8) sdata[tid] += sdata[tid+4];
    if (blockSize >=  4) sdata[tid] += sdata[tid+2];
    if (blockSize >=  2) sdata[tid] += sdata[tid+1];
  }

  //сохранить результат в глобальную память
  int out_idx = last? 0 : blockIdx.x;
  if (tid == 0) g_odata[out_idx] =  sdata[0];
}
```

Шаг параллельного проекционного метода Эйлера для CPU:

```
#pragma once
#include "StepSolverBase.h"
template <template<class R> class TBlas,class RealT,class Vector,
   class Func,class History>
struct PFEStep : public StepSolverBase<TBlas<RealT> >{
        typename StepSolverBase<TBlas<RealT> >::MyBlasVector
           inner_results;
        typename StepSolverBase<TBlas<RealT> >::MyBlasVector tmp;

        void init(unsigned int N,RealT * init){
                inner_results.reset(N+N);
                tmp.reset(N);
        }
       void call(unsigned int N,RealT t,RealT h, Vector &x, const
          Func &F,const History* history = 0)          {
```

```
            const int INNER_STEPS = 4;
            const RealT inner_h = h/16;
            RealT inner_t = t;
            for (unsigned int i=0;i<INNER_STEPS;++i){
                F(inner_t,x,tmp);
                TBlas<RealT>::axpy(N,inner_h,tmp,x);
                if(i==INNER_STEPS-1){

                        TBlas<RealT>::copy(N,x,&inner_results[N]);
                }
                if(i==INNER_STEPS-2){
                  TBlas<RealT>::copy(N,x,&inner_results[0]);
                }
               inner_t += inner_h;
            }
    const RealT psi = h/inner_h - INNER_STEPS;
          TBlas<RealT>::scal(N,psi+1,&inner_results[N]);
                                TBlas<RealT>::axpy(N,-psi,&
      inner_results[0],&inner_results[N]);
          TBlas<RealT>::copy(N,&inner_results[N],x);
        }
};
```

Исходный код функции шага проекционного метода Эйлера на GPU

```
__global__ void ode_step1_pfe1(RealT t,
RealT h,
RealT* x,
RealT *tmp,
int sz,
void* additional_data,
int i)
{
    //инициализация пропущена для краткости

    //интегрирование методом Эйлера
    F(t,x,tmp,sz,additional_data);
    axpy3(h,x,tmp,reduce_size);

    //Сохраняем предпоследнее значение внутреннего интегрирования x
    if(i==INNER_STEPS-2){
      int idx = compute_thread_index();
      data_gpu->inner_results[idx] = x[idx];
```

```
        idx+=reduce_size;
        data_gpu->inner_results[idx] = x[idx];
        idx+=reduce_size;
        data_gpu->inner_results[idx] = x[idx];
      }

      //на последнем шаге внутреннего интегрирования сразу выполняем
         шаг интерполяции
      if(i==INNER_STEPS-1)
      {
        const RealT inner_h = h/16;
        const RealT psi = h/inner_h - INNER_STEPS;
        int idx = compute_thread_index();
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) * x[
           idx];
        idx+=reduce_size;
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) * x[
           idx];
        idx+=reduce_size;
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) * x[
           idx];
      }

      //редукция, для корректной работы функции вычисления правой
         части
      __threadfence();
      __syncthreads();
      reduce_device(reduced_ptr,reduce_buffer,reduce_size);
}
```

Программа вычисления решения при помощи заданного метода (последовательный и параллельный варианты):

```
#pragma once

#include <boost/cstdint.hpp>
#include "BlasCommon.h"
#include "rk.h"
#include <omp.h>
#include <cassert>


template<class RealT,
```

```
        template<class Breal> class BlasT,
        template<template<class RealB> class Blas,class RealT,class
           Vector,class Func,class History> class TStepSolver,
        class FuncT,
        class VectorT
>
void solve_fixedstep(
                                        unsigned int N,
                                        RealT dbegin,
                                        RealT dend,
                                        RealT h,
                                        FuncT F,
                                        const VectorT& init,
                                        VectorT result)
{
        using boost::uint64_t;

        typedef BlasT<RealT> Blas;

        const uint64_t factor   =       1000000;
        const uint64_t ih               =       (uint64_t)(h*factor
           +0.5);
        uint64_t ibegin =       (uint64_t)(dbegin*factor+0.5);
        const uint64_t iend             =       (uint64_t)(dend*
           factor+0.5);

        VectorT &x = result;
        Blas::copy(N,init,x);

        typedef TStepSolver<BlasT,RealT,VectorT,FuncT,VectorDeque<
           BlasT<RealT>>> StepSolver;

        StepSolver solver;

        solver.init(N,init);

        //init history
        const unsigned int history_length = solver.history_length()
           ;
        VectorDeque<BlasT<RealT>> history(N,history_length - 1);
```

```
        if(history_length>1){   //requires at least 1 point of
           history, init with initial value
                history.push(init);
        }
        if(history_length>2){   //requires warmup
                const uint64_t warmup_end = ibegin+ih*(
                   history_length-1);
                for(uint64_t ti=ibegin+ih; ti<warmup_end; ti+=ih){
                        RealT t = ti/(RealT)factor;
                        Rk4Step<BlasT,RealT,VectorT,FuncT,
                           VectorDeque<BlasT<RealT>>>
                           warmup_solver;
                        warmup_solver.call(N,t,h,x,F,&history);
                        history.push(x);
                }
                ibegin = warmup_end - ih;
        }

        //main solver loop
        for(uint64_t ti=ibegin+ih;ti<iend;ti+=ih){
                RealT t = ti/(RealT)factor;
                solver.call(N,t,h,x,F,&history);
                if(history_length>1){
                        history.push(x);
                }
        }
}


template<class RealT,
        template<class Breal> class BlasT,
        template<template<class RealB> class Blas,class RealT,class
           Vector,class Func,class History> class TStepSolver,
        class FuncT,
        class VectorT
>
void solve_fixedstep_parallel(
                                        unsigned int N,
                                        RealT dbegin,
                                        RealT dend,
                                        RealT h,
```

```
                                        FuncT F,
                                        const VectorT& init,
                                        VectorT result,
                                        int nthreads = -1
                                        )
{
        using boost::uint64_t;

        typedef BlasT<RealT> Blas;

        const uint64_t factor   =       1000000;
        const uint64_t ih               =       (uint64_t)(h*factor
           +0.5);
        uint64_t ibegin =       (uint64_t)(dbegin*factor+0.5);
        const uint64_t iend             =       (uint64_t)(dend*
           factor+0.5);

        VectorT &x = result;
        Blas::copy(N, init, x);

        typedef TStepSolver<BlasT, RealT, VectorT, FuncT, VectorDeque<
           BlasT<RealT> > > StepSolver;

        StepSolver solver;

        solver.init(N, init);

        //main solver loop
        if(nthreads==-1)
                nthreads = 2;

#pragma omp parallel num_threads(nthreads)
        {
                const int thread_id = omp_get_thread_num();
                const int items_per_thread = N / nthreads;
                assert(N % nthreads == 0);
                const int chunk_begin = thread_id*items_per_thread;
                for(uint64_t ti=ibegin+ih;ti<iend;ti+=ih){
                        RealT t = ti/(RealT)factor;
                        solver.call(N,t,h,x,F,chunk_begin,
                           items_per_thread,0);
```

```
                        #pragma omp barrier
                        int braek = 0;
                }

        }
}
```

Реализация решения системы ОДУ на CUDA:

```
#include "../numeric\primitives\ker_blas.h"
#include "../numeric\primitives\refblas.h"
#include "../numeric\primitives\cudablas.h"
#include "../numeric\primitives\types.h"
#include "turb_equation_cuda.h"

#include <vector>
#include <iostream>
#include <boost/timer.hpp>

#include "ode_cuda.h"

typedef float RealT;
#define hostptr
#define deviceptr

#define THREADS_PER_BLOCK 128

__device__ void reduce_device(real_t* reduced_ptr, real_t*
   reduce_buffer, int reduce_size, bool print=false  )
{
        reduce2<real_t,THREADS_PER_BLOCK,false,false>(reduced_ptr,
           reduce_buffer,reduce_size);

        const unsigned int tid = threadIdx.x;
        __shared__ bool amLast;

        // wait until all outstanding memory instructions in this
           thread are finished
        __threadfence();

        // Thread 0 takes a ticket
        if( tid==0 )
        {
```

```
                unsigned int ticket = atomicInc(&retirementCount,
                    gridDim.x);

                // If the ticket ID is equal to the number of blocks
                    , we are the last block!
                amLast = (ticket == gridDim.x - 1);
        }
        __syncthreads();

        // The last block sums the results of all other blocks
        if(amLast) {
                reduce2<real_t,THREADS_PER_BLOCK,true,false>(
                    reduce_buffer,reduce_buffer,gridDim.x);

                if( tid==0 ){
                        // reset retirement count so that next run
                            succeeds
                        retirementCount = 0;
                }
        }
}


__device__ void F(RealT t,RealT*x,RealT* tmp,int sz,void*
    additional_data){
        DeviceTurbData* pdata = static_cast<DeviceTurbData*>(
            additional_data);
        TurbEquationCuda::turb_equation_main(t,x,tmp,sz,pdata);
}

__device__ void axpy3(RealT h, RealT* x, RealT *tmp, int sz)
{
        int i = compute_thread_index();
        x[i] = h*tmp[i] + x[i];
        //x[i] = tmp[i] ;
        i+=sz;
        x[i] = h*tmp[i] + x[i];
        i+=sz;
        x[i] = h*tmp[i] + x[i];
}
```

```
__global__ void ode_step1_euler(RealT t,RealT h, RealT* x, RealT *
   tmp,int sz,void* additional_data){
        F(t,x,tmp,sz,additional_data);

        sz/=3;

        axpy3(h,x,tmp,sz);

        __threadfence();
        __syncthreads();

        DeviceTurbData* data_gpu = (DeviceTurbData*)additional_data;
        float* reduced_ptr = x;
        float* reduce_buffer = data_gpu->reduce_buffer;
        const int reduce_size = sz;

        reduce_device(reduced_ptr,reduce_buffer,reduce_size);
}

__global__ void ode_step1_pfe1(RealT t,RealT h, RealT* x, RealT *tmp
   ,int sz,void* additional_data, int i)
{

                const int INNER_STEPS = 4;
                DeviceTurbData* data_gpu = (DeviceTurbData*)
                   additional_data;
                float* reduced_ptr = x;
                float* reduce_buffer = data_gpu->reduce_buffer;
                const int reduce_size = sz/3;

                F(t,x,tmp,sz,additional_data);

                axpy3(h,x,tmp,reduce_size);

                if(i==INNER_STEPS-2){
                        int idx = compute_thread_index();
                        data_gpu->inner_results[idx] = x[idx];
                        idx+=reduce_size;
                        data_gpu->inner_results[idx] = x[idx];
                        idx+=reduce_size;
                        data_gpu->inner_results[idx] = x[idx];
```

```
                }

                if(i==INNER_STEPS-1)
                {
                        const RealT inner_h = h/16;
                        const RealT psi = h/inner_h - INNER_STEPS;
                        int idx = compute_thread_index();

                        x[idx] = -psi * data_gpu->inner_results[idx]
                            + (psi + 1) * x[idx];
                        idx+=reduce_size;
                        x[idx] = -psi * data_gpu->inner_results[idx]
                            + (psi + 1) * x[idx];
                        idx+=reduce_size;
                        x[idx] = -psi * data_gpu->inner_results[idx]
                            + (psi + 1) * x[idx];

                }

                __threadfence();
                __syncthreads();
                reduce_device(reduced_ptr,reduce_buffer,reduce_size)
                    ;
}

__global__ void ode_step1_pfe2(RealT h, RealT* x, int sz, void*
    additional_data)
{
        const int INNER_STEPS = 4;
        const RealT inner_h = h/16;
        DeviceTurbData* data_gpu = (DeviceTurbData*)additional_data;
        float* reduced_ptr = x;
        float* reduce_buffer = data_gpu->reduce_buffer;
        const int reduce_size = sz/3;

        const RealT psi = h/inner_h - INNER_STEPS;

        int idx = compute_thread_index();
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) *
            data_gpu->inner_results[sz+idx];
        idx+=reduce_size;
```

```
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) *
            data_gpu->inner_results[sz+idx];
        idx+=reduce_size;
        x[idx] = -psi * data_gpu->inner_results[idx] + (psi + 1) *
            data_gpu->inner_results[sz+idx];

        __threadfence();
        __syncthreads();

        reduce_device(reduced_ptr,reduce_buffer,reduce_size,true);

}

__global__ void ode_step1_reduce(RealT* reduced_ptr, int sz, void*
    additional_data)
{
        DeviceTurbData* data_gpu = (DeviceTurbData*)additional_data;
        float* reduce_buffer = data_gpu->reduce_buffer;
        const int reduce_size = sz/3;

        reduce_device(reduced_ptr,reduce_buffer,reduce_size);
}



void run_ode_step(RealT t,RealT h, RealT* x,int sz, RealT* tmp,void*
     additional_data){
        dim3 dimBlock(THREADS_PER_BLOCK,1,1);
        dim3 dimGrid(sz/3/THREADS_PER_BLOCK,1,1);
        ode_step1_euler<<<dimGrid,dimBlock >>>( t, h,  x, tmp , sz,
            additional_data);
}

void run_ode_step_pfe(RealT t,RealT h, RealT* x,int sz, RealT* tmp,
    void* additional_data){
        dim3 dimBlock(THREADS_PER_BLOCK,1,1);
        dim3 dimGrid(sz/3/THREADS_PER_BLOCK,1,1);

        const int INNER_STEPS = 4;
        const RealT inner_h = h/16;
        RealT inner_t = t;
```

```
        for (unsigned int i=0;i<INNER_STEPS;++i)  {
                ode_step1_pfe1<<<dimGrid,dimBlock >>>( inner_t,
                   inner_h,  x, tmp , sz,additional_data,i);
                inner_t += inner_h;
        }
}



void solve_fixedstep_gpu(
                                        unsigned int N,
                                        RealT dbegin,
                                        RealT dend,
                                        RealT h,
                                        const RealT* init,      //
                                           gpu pointer
                                        RealT* result,
                                                          //gpu
                                           pointer
                                        void * additional_data //
                                           gpu pointer
                                        )
{
        typedef unsigned __int64 uint64_t;

        typedef CUDABlas Blas;

        const uint64_t factor   =       1000000;
        const uint64_t ih               =       (uint64_t)(h*factor
           +0.5);
        uint64_t ibegin =       (uint64_t)(dbegin*factor+0.5);
        const uint64_t iend             =       (uint64_t)(dend*
           factor+0.5);

        const int nsteps=3;

        RealT* &x = result;
        Blas::copy(N,init,x);

        LOG_GPU(init);
        LOG_GPU(x);
```

```
        BlasVector<CUDABlas> tmp(N*3);
        BlasVector<RefBlas<float> > x_cpu(N*3);
        //main solver loop
        {
                for(uint64_t ti=ibegin+ih;ti<iend;ti+=ih){

                        RealT t = ti/(RealT)factor;
                        run_ode_step_pfe(t,h,x,N,tmp,additional_data
                            );
                        //run_ode_step(t,h,x,N,tmp,additional_data);
                        int braek = 0;
                }

        }
        //cudaDeviceReset();

}

void dosolve3(TurbSystem_dataT<RefBlas<real_t> >& data){

        cublasInit();

        TurbEquationCuda teq;
        teq.test_load(data);

        BlasVector<CUDABlas> init;

        teq.fill_init_val(init);

        BlasVector<CUDABlas> result(teq.data().N*3);

        const int nequations = teq.data().N*3 ;
        boost::timer timer;

        real_t beg1 = data.dbegin;
        real_t end1 = data.dend;

        DeviceTurbData *data_gpu;
        cudaMalloc((void**)&data_gpu,sizeof(*data_gpu));
        cudaMemcpy((void*)data_gpu,(void*)&teq.data_main,sizeof(*
            data_gpu),cudaMemcpyHostToDevice);
```

```
        solve_fixedstep_gpu(nequations,
                beg1,end1+data.h/2,
                data.h,
                &init[0],
                &result[0],
                data_gpu
        );

        std::vector<real_t> result_cpu(teq.data().N*3);
        CUDABlas::extract(teq.data().N*3,&result[0],&result_cpu[0]);

        cudaDeviceReset();
}
```